QBasic-Kochbuch V1.0

http://kickme.to/tiger/

```
******************************************************************************
******************************************************************************
**
**                           Das QBasic 1.1 Kochbuch
**                        =============================
**
**  Eine Bauanleitung für QBasic-Programme mit Hinweisen auf die Unterschiede
**  zu MS QuickBasic und PowerBasic.
**
**  Dieses QBasic-Kochbuch wird permanent verbessert und erweitert. Die neueste
**  Version und die zitierten Beispielprogramme können beim Verfasser als Off-
**  line-Version per E-Mail unter thomas.antoni@erlf.siemens.de
**  angefordert werden. Verbesserungsvorschläge sind stets willkommen.
**
**           (c) Thomas Antoni, Version 1.0, 07.03.99 - 21.12.99
**
******************************************************************************
******************************************************************************
```

```
******************************************************************************
```
* **Inhalt**

```
******************************************************************************
```

** zum Inhalt ***

* **Vorwort**

****

Dieses QBasic-Kochbuch ist eine 'Bauanleitung' für QBasic-Programme. Für alle wichtigen Programmierprobleme werden in übersichtlicher Form die benötigten QBasic-Befehle genannt und erläutert - illustriert durch eine Fülle von Beispielen. Das QBasic-Kochbuch ist gleichzeitig eine Kurzreferenz fast aller QBasic-Befehle und soll die sehr gute Online-Hilfe von QBasic 1.1 ergänzen. Das QBasic-Kochbuch ersetzt jedoch nicht einen Einführungskurs in QBasic; Neueinsteigern sei hierfür die hervorragenden Tutorials {2}, {1} und {4} empfohlen.

QBasic ist ein reiner Interpreter und kostenlose Beigabe von MS-DOS ab V5.0 sowie Windows 95/98. Umsteiger auf die echten Compiler QuickBasic und PowerBasic (ehemals Borland TurboBasic) erhalten die wichtigsten Informationen für den Umstieg.

Die Literaturhinweise beziehen sich auf die am Schluß aufgelisteten Bücher und sind wie folgt aufgebaut: {x/n} = Seitennummer n im Buch {x}.

Credits: Dank an Hawkynt für seine Ergänzungen und Korrekturen zum Abschnitt
"Umstieg von QBasic nach PowerBasic V3.5"

** zum Inhalt ***

* **Online-Hilfe verwenden**

****

- Rechter Mausklick auf ein Schlüsselwort im Programmlisting oder im Hilfefenster öffnet die Hilfe zum Schlüsselwort.
  Bei PowerBasic ist die kontextsensitive Hilfe zu dem Schlüsselwort, auf dem der Cursor steht, über <Strg + F1> erreichbar.
- Im Hilfefenster kann mit <Bearbeiten> <Suchen> über alle Hilfethemen hinweg nach Text gesucht werden (sehr nützlich!).
- Hilfe beenden mit Esc-Taste

** zum Inhalt ***

* **Bedienung und Aufruf von Editor, Interpreter, Compiler und EXE-Programmen**
*   {11/25+456}

****

```
- Programm während Interpretation abbrechen:  <Strg+Pause> (auch bei Absturz)
          (evtl.auch <Break> oder  <Ctrl+C> )
- Aufruf eines Programms *.BAS von DOS aus:
          QBASIC /RUN *.BAS    'das Programm kehrt zu DOS zurück, wenn es mit
                               'SYSTEM statt END abgeschlossen ist
- QBASIC /H - startet QBasic mit der vollen VGA-Auflösung von 50 Zeilen
          (QB /H bei QuickBasic)
- QBASIC /EDITOR [/H] <Filename> - startet den in QBASIC enthaltenen MS-DOS-
          Texteditor [im 50-zeiligen VGA-Modus]; Texteditor nur in QBasic (max.
```

Dateilänge 64 K), nicht in QuickBASIC vorhanden!

- Debug-Funktionen: {7/19} {11/459}
    - Haltepunkte setzen/ rücksetzen: <Cursor auf den gewünschten Befehl setzen>, dann <Debug|Haltepunkt ein/aus> oder [F9]
    - Am Haltepunkt Variablen anschauen: Ist im "Direkt"-Fenster durch die Eingabe von PRINT <Variable> möglich. Ausgabebildschirm über <Ansicht | Ausgabebildschirm> oder [F4]-Taste kontrollierbar. Bei QuickBASIC u. PowerBASIC ist die Variablenanzeige komfortabler möglich über <Debug|Variable anzeigen> bzw. <Break/Watch|Add Watch> {7/20}
    - Programme in Zeitlupe ablaufen lassen mit Hervorhebung des jeweils aktuellen Befehls und kurzem Halt an den Schleifenanfängen: <Debug|Rückverfolgung ein> <Ausführen|Start> oder den fraglichen Programmbereich durch die Befehle TRON und TROFF einklammern (Trace On/Off). Die Zeitlupe ist nicht immer praktikabel, besonders im Windows-DOS-Fenster.
    - Vom aktuellen Befehl aus im Einzelschritt Befehl für Befehl ausführen: <Debug|Einzelschritt> oder [F8]. Sollen SUBs und FUNCTIONs normal (ohne Einzelschritt) durchlaufen werden, so erfolgt die Schrittfortschaltung über <Debug|Prozedurschritt> oder [F10].
    - Aktuellen Befehl festlegen - zum Überspringen eines Programmabschnitts: <Cursor auf den gewünschten Befehl setzen> <Debug | nächste Anweisung festlegen>
- Anzeige von Unterprogrammen mit [F2]
  Durchblättern der Unterprogramme mit [Shift+F2]
- Ein zweites Editierfenster kann mit <Ansicht|Aufteilen> geöffnet werden, z.B. um Hauptprogramm und Subroutine gemeinsam auf dem Bildschirm zu haben. {6/88}
- Max. Länge von Code und Daten: Insgesamt 160K (bei QuickBasic und PowerBasic nur durch den freien Speicherplatz im konventionellen unteren 640K-Speicher begrenzt).
- Library-Funktion (nur bei QuickBasic und PowerBasic): Über 'QB /L' läßt sich QuickBasic mit der Qick-Library QB.QLB aufrufen: (erforderlich z.B. bei Verwendung der Befehle INTERRUPT[X], CALL ABSOLUTE ...) {9/6}
- Bei QuickBasic lassen sich die Pfade zu den Include-, Lib- usw. -Files individuell setzen {9/2}
- Parameterübergabe von der Aufruf-Kommandozeile an ein kompiliertes QuickBasic-Programm: Das Programm kann die übergebenen Aufrufparameter über COMMAND$ abfragen (nur von QuickBasic, nur von QuickBasic unterstützt; siehe auch SUB GetArguments in QSUBFUN.BAS)
- Compiler und Linker sind bei QuickBasic und PowerBasic auch als eigenständige EXE-Programme vorhanden, um mehr Arbeitsspeicher zum Erzeugen großer Programme zur Verfügung zu haben.

** zum Inhalt ***

* Syntax

****

- 1 Befehl darf nicht länger als 1 Zeile sein
- In eine Zeile darf man mehrere Befehle schreiben, jeweils durch ':' voneinander getrennt.
- Kommentar hinter ' oder REM einfügbar (REM nur am Zeilenanfang)
- Sprungmarken <Zahl> oder '<Name>:', z.B. '120' oder 'Start:'
- Programm wird beendet mit den Befehlen 'END' oder 'SYSTEM'. SYSTEM bewirkt Aussprung zu DOS aus dem Interpreter heraus, wenn das Programm vom außerhakb des QBasic-Interpretes mit QBASIC /RUN prog.BAS aufgerufen wurde. END bewirkt einen Rücksprung zum QBASIC-Interpeter/Editor bewirkt.

- Namen von Variablen, Subroutinen und Funktionen: : Max 40 Zeichen, nur Buchstaben, Ziffern und Punkt, 1.Zeichen=Buchstabe (Beispiel: 'Parfum.4711'). Groß-/Kleinschreibung wird nicht unterschieden.

** zum Inhalt ***

* **Variablen**

****

- Datentypen (durch nachfolgendes Typendefinitionszeichen (Suffix) gekennzeichnet, Deklaration nicht erforderlich, aber mit DIM möglich, s.u.):
    - anna% : INTEGER, Ganzzahl mit Vorzeichen (16 bit) -32768...32767
    - otto& : LONG, lange Ganzzahl mit Vorzeichen (32 bit) -2147483648...2147483647
    - egon! : SINGLE, einfach lange Gleitpunktzahl (32 Bit, 7 Stellen genau) +-2,802597 *10^-45...+-3,402823 *10^38
    - egon : dito (Suffix "!" ist bei SINGLE-Variablen weglaßbar)
    - paul# : DOUBLE, doppelt lange Gleitpunktzahl (64 Bit, 16 Stellen genau) +-4,446590812571219 *10^-323...+-1,79769313486231 *10^308
    - duda$: STRING, Text-String (Zeichenkette, max. ca. 32767 Zeichen)

  Negative Zahlen werden im Zweierkomplement dargestellt (-1 = FFFFhex).
- Der verfügbare Speicherplatz für Variablen, und Stack läßt sich mit FRE und CLEAR abfragen und vergrößern; siehe Absatz 'Vorhandenen freien Speicherplatz für Variablen...' im Abschnitt 'Direkter Speicherzugriff...' .
- Zusätzliche Datentypen bei PowerBasic (die Länge von Strings ist nur durch den freien Speicherplatz begrenzt {11/486}):
    - hugo&& : QUAD, vierfach lange Ganzzahl mit Vorzeichen (64 Bit)
    - hugo## : EXTENDED, Riesen-Gleitpunktzahl (80 Bit)
    - hugo@ : FIX, BCD-Festpunktzahl
    - hugo@@ : BCD, BCD-Gleitpunktzahl
    - hugo$$: FLEX Flexibler String (max 32750 Zeichen)
- Datendeklarationen (am Programmanfang; nicht erforderlich, aber nützlich für den automatischen Typ-Check)
    - DIM [SHARED] <Variablenname ohne Suffix> AS <Typ> [,<Variable 2> AS..]
        - SHARED ==> auch Subroutinen können die Variable verwenden (Ohne SHARED sind sie dort unbekannt!). Umgekehrt kann das Hauptprogramm nur auf Variable einer Subroutine zugreifen, die dort mit SHARED deklariert sind. Bei PowerBasic kann DIM bei Verwendung von SHARED entfallen.
        - <Typ> = INTEGER|LONG|SINGLE|DOUBLE|STRING -(Datentyp, s.o.)
        - Beispiele: DIM anna AS LONG, otto AS SINGLE : anna& = 1.234
          DIM text AS STRING [*12] - [statischer String mit der festen Länge von 12 Zeichen]
        - Bei der nachfolgenden Verwendung einer mit DIM deklarierten Variablen kann der Typ-Suffix weggelassen werden
    - Variable, auf die das Hauptprogramm und ein Unterprogramm zugreifen können, müssen mit COMMON und SHARED als Globalvariable deklariert werden; siehe Abschnitt 'Geltungsbereich der Variablen')
- Standard-Datentypen für Variable und FUNCTIONen festlegen:
    - {DEFINT|DEFLNG|DEFSNG|DEFDBL|DEFSTR} <Buchstabenliste> : Alle nicht über einen Datentyp-Suffix (%, &, !, #, or $) definierten Variablen mit dem Anfangs<Buchstaben> werden auf den Typ INTEGER|LONG|SINGLE|DOUBLE|STRING gesetzt. Bei der nachfolgenden Verwendung der Variablen kann der Suffix weggelassen werden. DEFxxx-Anweisungen müssen in SUBs und FUNCTIONs wiederholt werden; QBasic fügt dies automatisch ein.

```
                Beispiel: DEFINT A-Z ==> alle Variablen ohne Suffix sind auto-
                                   matisch vom Typ INTEGER (siehe NIBBLES.BAS)
- Typenumwandlungen
    - implizit: Bei einer impliziten Gleitpunkt ==> Integer Wandlung wird
                auf die nächstgelegene Ganzahl gerundet; Sonderfall: xxx.5
                wird auf die nächste gerade Zahl gerundet. Beispiele:
                otto% = 5.5                  ==> otto% := 6 (Sonderfall)
                otto% = 6.5                  ==> otto% := 6 (Sonderfall)
                otto% = 2.678                ==> otto% := 3
                otto% = 23.42                ==> otto% := 23
                anna! = 3.51 : otto% = anna! ==> otto% :=  4
    - explizit: FIX (<Ausdruck>) - erzeugt den ganzzahligen Anteil d.Ausdrucks
                     durch Abschneiden der Nachkommastellen. Es wird nicht
                     gerundet im Gegensatz zur impliziten Typenumwandlung.
                     Beispiele: FIX(12.45) ==> 12; FIX(-12.89) ==> -12
                INT (<Ausdruck>) - gibt die größte Ganzzahl zurück, die kleiner
                     oder gleich dem Ausdruck ist. Mit INT kann man auch Run-
                     dungen aller Art durchführen; siehe Abschnitt
                     'Arithmetische Operatoren...'.
                     Beispiele: INT(12.45) ==> 12; INT(-12.89) ==> -13
                CDBL <Ausdruck>  - numer. Ausdruck in DOUBLE-Gleitpunktzahl um-
                                   wandeln (kann keinen numer. Stringausdruck
                                   (z.B. "2*3") konvertieren!)
                CSGN <Ausdruck>  - numer. Ausdruck in SINGLE-Gleitpunktzahl
                                   umwandeln
                CINT <Ausdruck>  - Runden auf eine INTEGER-Ganzzahl
                CLNG <Ausdruck>  - Runden auf eine LONG-Ganzzahl
                VAL  (<String>)  - String in Zahl umwandeln, z.B. VAL("2.34")
                STR$ (<Zahl>)    - Zahl in String umwandeln, z.B. STR$(2.34)
                CQUD | CEXT | CFIX | CBCD - zusätzliche Typwandlungen für die
                                   speziellen PowerBasic-Datentypen

***************************************************************** zum Inhalt ***
* Felder        {1/57}  {3/12f}  {6/102}
********************************************************************************
- Deklaration: DIM [SHARED] <Feldname> (<Anzahl Feldelemente%-1>) - z.B.
                DIM player$(3) - hat 4 Feldelemente: player$(0)...(3)
                [SHARED] ==> Feld auch von SUBs u. FUNCTIONs ansprechbar
        oder : DIM <Name> (Nr.des ersten Elements> TO <Nr. des letzten
                Elements> - z.B. DIM player$(1 TO 4). Hinweis: Bei PowerBasic
                'TO' durch ':' ersetzen!
- OPTION BASE  <n%> - Verschiebt die Feldelement-Nummern aller Felder des Pro-
        gramms um n% {6/207}. Normale Verwendung: n%=1; Beispiel:
        OPTION BASE 1: DIM anna%(5) ==>Indices laufen von 1...5 statt von 0...4
- Wertzuweisung: <Feldname> (<Nr.des Feldelements%>) = <Wert>
        z.B. player$(1) = "Tom"
- Feld zurücksetzen: ERASE <Feldname>1> [,<Feldname2>...] ;numerische Fel-
        der werden auf 0, Stringfelder auf "" gesetzt. Feld bleibt in voller
        Länge erhalten - außer bei dynamischen Feldern (s.u.)
- Max Feldlänge: Integer: 2^16-2 = 65534 Bytes, Long Integer: 2^16 Bytes
- LBOUND (<Feldname> [,Dimension%]) - liefert die kleinste Feldelement-Nr. (den
        kleinsten Index) des Feldes zurück (untere Grenze); bei mehrdimensio-
        nalen Felder die [Dimension] angeben. LBOUND wird z.B. von Subroutinen
        benötigt, die beliebige Felder bearbeiten sollen (siehe SORT.BAS).
- UBOUND (<Feldname> [,Dimension%]) - liefert den größten Index des
```

Feldes zurück (obere Grenze)
- Deklaration eines mehrdimensionalen Feldes mit einheitlichen Datentypen:
```
DIM anna% (1 TO 10, 1 TO 8)  ' Deklaration (in PowerBasic: (1:10, 1:8))
anna% (3, 6) = otto%         ' Wertzuweisung
```
- Deklaration eines mehrdimensionalen Verbund-Feldes gemischten Typs ("Anwenderdefinierter Typ"); muß im Hauptprogramm, darf nicht in SUB oder FUNCTION deklariert werden {11/269}:
```
TYPE quiz                    'Datentyp "quiz" deklarieren: Feld m. je
  frage AS STRING * 70       '3 String-Elementen (70, 50 und 50 Zei-
  antw1 AS STRING * 50       'chen lang) und einem Integer-Element
  antw2 AS STRING * 50       'Typ-Schlüsselwörter (STRING, INTEGER..):
  oknr  AS INTEGER           'siehe Abschnitt 'Variablen'
END TYPE

DIM geschichte (1 TO 20) AS quiz  'Anwenderdefiniertes Feld vom Typ
   '"quiz" deklarieren (auch in SUB oder FUNCTION möglich). Die Dimen-
   'sionierung kann auch dynamisch erfolgen (z.B. '1 TO x%') {11/271}.
geschichte (1).frage = "Wer war der 1.Kanzler"      'Wertzuweisung
geschichte(1).antw1  = "Erhard"                      'Suffix weg-
geschichte(1).antw2  = "Adenauer"                    'lassen
geschichte(1).oknr = 2

DIM puffer (2) AS quiz       'Wertzuweisung 'en bloc für ein gesamtes
puffer(1) = geschichte(1)    'Verbund-Feldelement ist auch möglich
                             '(großer Vorteil!!!)
```
Hinweis 1: In anderen Programmiersprachen , z.B. C++, werden die anwenderdefinierten Felder oft auch "Strukturen" genannt.

Hinweis 2: PowerBasic kennt Verbundfelder erst ab V3.5! Diese lassen sich u.U. durch Flex-Strings nachbilden; numerische Größen müssen hierbei durch STR$|VAL in Strings gewandelt|rückgewandelt werden.

- Deklaration eines dynamischen Feldes (Feldlänge läßt sich zur Laufzeit verändern, Feld läßt sich wieder aus dem Speicher entfernen) {9/64+151} {6/215} {11/265}:
  - Variante 1:
```
DIM DYNAMIC feld%(15)   'Feld konstanter Länge deklarieren
ERASE feld%             'Feld aus dem Speicher entfernen
```
  - Variante 2:
```
INPUT n%
DIM feld% (n%):...  'Feld variabler Länge deklarieren;Feldlän-
INPUT m%            'ge ändern und Feld initialisieren ==> alte
REDIM feld%(m%)     'Daten werden gelöscht {11/265}, evtl. vor-
                    'her in Hilfsfeld retten
ERASE feld%         'Feld aus dem Speicher entfernen
```
  - '$STATIC | '$DYNAMIC - über diese 'Metabefehle' läßt sich voreinstellen, ob alle nachfolgenden per DIM deklarierten Felder statisch oder dynamisch angelegt werden sollen (weniger gebräuchliche Befehle {11/268})
- Übergabe von Feldern an SUBs und FUNCTIONs: Felder können als Parameter an die SUB oder FUNCTION übergeben werden (mit leeren Klammern ()). Sie müssen dort jedoch erneut deklariert werden {9/98} {6/223}; siehe Abschnitt 'Allgemeines zu Subroutinen...' . Dort ist auch die Übergabe anwenderdefinierter Verbundfelder beschrieben.
- Hinweis zu PowerBasic: Dort lassen sich über ARRAY {SORT | SCAN | INSERT | DELETE} Feldelemente komfortabel sortieren, suchen, einfügen und löschen.

** zum Inhalt ***

* **Konstanten**

```
******************************************************************************
CONST  <Konstantenname> = Ausdruck [,<Konstantenname> = Ausdruck]... - Deklara-
       tion von Konstanten, z.B. CONST pi# = 3.14159265358979  {3/55}
       Die CONST-Deklaration muß vor Verwendung der Konstanten erfolgen!
       Bei PowerBasic sind nur INTEGER-Ganzzahlkonstanten möglich (CONST durch
       '%' ersetzen, z.B. %anzahl = 37 statt CONST anzahl% = 37)
- DATA <Konstante1> [<Konstante2>,...] - Deklaration von Konstanten, die mit
       dem READ-Befehl eingelesen werden können  {2/14} {6/194}.
       Beispiel:   READ a%, text$    ==> a% = 2411;  text$ = "otto"
                   ...
                   DATA 2411, "otto"
       Data-Befehle können an beliebiger Stelle vor oder hinter dem READ-Befehl
       stehen, nicht jedoch ein einer SUB oder FUNCTION.
- RESTORE - ermöglicht ein Wiederaufsetzen auf die erste per DATA deklarierte
       Konstante (mehrmaliges Verwenden der DATA-Werte  {9/25f}.
       Zeilenweises RESTORE <Marke$> ist auch möglich, wenn die DATA-Zeile
       mit einer <Marke$:> versehen ist; siehe NIBBLES.BAS und {5/57}
- Beispiele für Konstanten:
   > -2.37        (SINGLE-Gleitpunktzahl)  > "Egon" (Text-Zeichenkette, String)
   > 235.988 E-7  (= 0.0000235988; SINGLE) > &H5AB  (Hexa-Zahl  5AB)
   > -2.5 D100    (DOUBLE-Gleitpunktzahl)  > &o173  (Oktal-Zahl 173)
                                           > Dualzahl nicht vorgesehen !!

**************************************************************** zum Inhalt ***
```

*** Wertzuweisungen, Ausdrücke, Operatoren**

```
******************************************************************************
Wertzuweisungen
---------------
- LET-Befehl: Der bei anderen Basic-Dialekten für Wertzuweisungen erforderliche
    LET-Befehl ist möglich, aber nicht zwingend, z.B. ist x=1 identisch mit
    LET x=1 .
- SWAP <Variable 1>, <Variable 2> - Der Wert beider Variablen wird vertauscht
- QBASIC initialisiert bei Programmaufruf alle numerischen Variablen mit dem
  Startwert '0' und alle Strings mit ""  {9/31}
- ERASE <feldname$) - setzt Felder auf den Startwert '0' bzw "" zurück (siehe
    Abschnitt 'Felder').
- CLEAR - inititialisiert alle Variablen mit dem Startwert '0' bzw ""  {11/251}

Arithmetische Operatoren und Funktionen {3/115ff}   (Priorität: siehe unten)
--------------------------------------------------
-  + - * /     - Grundrechenarten (Division durch '0' führt zum Fehlerabbruch!)
- ^            - Exponentialzeichen, z.B.        2^10 ==> 1024
                                              2^(1/3) ==> 3.Wurzel aus 2
- \            - Ganzzahl-Division, schneidet den Rest ab,
                  z.B. 19\7 ==> 2 ;  -19\7 ==> -2 ;  25.68\6.99 ==> 3
- x MOD y      - Dividiert x durch y und gibt den Rest als Ganzzahl zurück
                 (ist x oder y eine Gleitkommazahl, so wird sie vorher gerundet)
                       z.B.   19 MOD 7 ==> 5  ;   10.4 MOD 4 ==> 2
                            19 MOD 6.7 ==> 5  ;-17.6 MOD 3.7 ==> -2
                Der rückgelieferte Wert hat das gleiche Vorzeichen wie x
- SGN(x)       - Vorzeichen von x, liefert -1|0|1, wenn x kleiner|gleich|größer
                Null ist
- ABS (x)      - Absoluter Betrag einer Zahl, z.B. ABS (-82) ==> 82
- INT (x!)     - Integerwert erzeugen; liefert die nächstkleinere ganze Zahl,
                  z.B.      INT (2.79) ==> 2  ;  INT (-2.79) ==> -3
```

```
- INT (x+0.5) - x! auf die nächstgelegene ganze Zahl auf-/abrunden, z.B.
                INT (5.67 + 0.5)==> 6.
                Eine Rundung von Gleitpunktzahlen auf eine beliebige Anzahl
                von Vor- und Nachkommastellen ist folgendermaßen möglich:
                  - INT (x * 10^n + 0.5) / 10^n ==> x auf n Nachkommastellen
                    auf-/ abrunden (bzw. n Vorkommastellen, wenn n nega-
                    tiv ist)   {9/84} {3/127}
                  - Beispiel 1: auf 3 Nachkommastellen runden:
                    INT (1.23456 * 10^3 + 0.5) / 10^3    ==> 1.235
                  - Beispiel 2: auf ganze Hunderter runden:
                    INT (320.5 * 10 ^-2 + 0.5) / 10 ^-2 ==> 300
- FIX (x!)    - Nachkommastellen abtrennen ohne Rundung
                 z.B.      INT (2.79) ==> 2  ;  INT (-2.79) ==> -2
- SIN|COS (x) - Trigonometrische Funktionen Sinus und Cosinus {3/116} {11/322}
                (Winkel in Bogenmaß (Radian) eingeben: 2*pi# entspr. 360°)
                   z.B. cos 60° ausrechnen      ==>
                        COS(60 * 3.14159265358979 / 180) ==> 0.5
                Ergebnis ist vom Typ DOUBLE.
- TAN|ATN (x) - Tangens und Arcustangens
- EXP (x)     - x-te Potenz zur Basis e (e^x); Ergebnis ist vom Typ SINGLE
- LOG (x)     - Logarithmus zur Basis e (Natürlicher Logarithmus);
                Ergebnis ist vom Typ SINGLE {11/326}
- SQR (x)     - Quadratwurzel (Ergebnis vom Typ SINGLE); negative Zahlen x
                führen zu Fehlerabbruch; Kubikwurzel: Siehe oben bei "^"
- STR$ (x)    - Zahl  x in Zeichenkette umwandeln (ASCII-String)   {3/120}
                   z.B. STR$(10.234) ==>  " 10.234"
- VAL (<String>) - String in Zahl umwandeln, z.B. VAL ("2.43") ==> 2.43
- HEX$ (x)    - Zahl in Hexadezimalzahl-Zeichenkette umwandeln
                   (z.B. HEX$(100) ==> "64" = Hexadezimalwert von 100)
- folgende arithmetische Funktionen lassen sich durch Kombination der obigen
  Grundoperationen realisieren {9/87}:         |  CONST pi#  = 3.14159265358979
    ArcSin (x) =  ATN(x/SQR(-x*x + 1))         |  Cot (x)    = 1 / TAN (x)
    ArcCos (x) = -ATN(x/SQR(-x*x + 1))+pi#/2   |  ArcCot (x) = - ATN(x) + pi#/2
    Loga (a,x) = LOG (x) / LOG (a)  'Logarithmus von x zur Basis a
    NteWurzel (n,x) = x ^ (1/n)     'N-te Wurzel aus x

Vergleichsoperatoren  (Ergebnis = -1|0, wenn Bedingung erfüllt|nicht erfüllt)
--------------------  (Vergleich von Texten: siehe {9/57} )
-   =  | <  | >   - gleich   | größer               | kleiner
-   <> | <= | >=  - ungleich | kleiner oder gleich | größer oder gleich

Logikoperatoren  (auf INTEGER- und LONG-Integer-Zahlen anwendbar) {3/102}{1/125}
-----------------------------------------------------------------
- NOT x%      - Logische Invertierung (bitweises Komplement)
- x% AND y%   - Verundung (bitweise Konjunktion)
- x% OR  y%   - Veroderung (bitweise Disjunktion)
- x% XOR y%   - Exklusiv-Oder-Verknüpfung (bitweise Antivalenz)
- x% EQU y%   - bitweise Äquivalenz (Ergebnis jew.=1, wenn beide Bits gleich)
                  (Umkehrung der XOR-Funktion, wenig gebräuchlich)
- x% IMP y%   - bitweise Implikation (=NOT(x% AND (NOT y%)); wenig gebräuchl.)


Operatoren und Funktionen für Zeichenketten (Strings):
-----------------------------------------------------
{3/108 f} {2/16f} {9/67} {6/249ff}
```

```
- LEN    (<String>)     - Länge des Strings (Typ Integer)
- LEFT$  (<String>, <Anzahl zu extrahierende Zeichen>)
                        - String v.links zerlegen (Anzahl >= 0)
- RIGHT$ (<String>, <Anzahl zu extrahierende Zeichen>)
                        - String v. rechts zerlegen (Anzahl >= 0)
- MID$   (<String> , <Anfang> [, <Anzahl zu extrahierenden Zeichen>] )
                        - Teilstring ermitteln; <Anfang>= Nummer d.1. zu extra-
                          hierenden Zeichens; 1.Stringzeichen hat die Nummer "1"
                          Bei weggelassener [,<Anzahl..>] werden alle Zeichen des
                          String ab <Anfang> extrahiert. Anzahl... >= 0.
- LCASE$ (<String>)     - Kleinbuchstaben erzeugen (versagt bei Umlauten){11/335}
- UCASE$ (<String>)     - Großbuchstaben erzeugen (versagt bei Umlauten);
                          z.B. Eingabe unabhängig von Groß-/Kleinschreibung ge-
                          stalten:
                             INPUT Text$: IF UCASE$(Text$) = "END" THEN END
- LTRIM$ (<String>)     - Führende Leerzeichen entfernen
- RTRIM$ (<String>)     - nachfolgende Leerzeichen entfernen
- SPACE$ (15)           - String mit 15 Leerzeichen erzeugen
- VAL    (" 3.1E12cm")  - String in eine Zahl umwandeln ==> 3.1 E+12
                          Führende und nachfolgende Blanks und Buchstaben werden
                          überlesen und stören nicht.  {3/112} {9/67}
- STR$   (<Zahl>)       - Zahl in String umwandeln {3/120} {7/62} {11/209}
- HEX$   (<Zahl>)       - Zahl in Hexa-Zahl-Zeichenkette umwandeln,
                          z.B. x = 100 : t$ = HEX$(x) ==> t$ = "64"
- OCT$   (<Zahl>)       - Zahl in Oktal-Zahl-Zeichenkette umwandeln
- INSTR ([<Beginn%>,] <String1> <String2>)
                        - String suchen {2/19}: sucht ab dem ersten [bzw. dem
                          <Beginn%>-ten]Zeichen in String1 nach String2 und gibt
                          die Zeichenposition des ersten Auftretens zurück bzw.
                          0, wenn String 2 nicht gefunden wird  {7/19}.
- STRING$ (<n>, <String>)
                        - gibt eine Zeichenfolge zurück, die <n>-mal hinter-
                          einander das erste Zeichen des <String> enthält,
                          z.B.: PRINT STRING$(80, "-") ==> Strich ziehen
- <Text1> + <Text2>     - Aneinanderfügen mehrerer Texte zu einem String
- ASC (<String>)        - gibt den ASCII-Code des ersten Stringzeichens zurück;
                          z.B.  ASC("ABCD")  ==>  65   (ASCII-Code von "A")
- CHR$ (<Code%>)        - gibt das ASCII-Zeichen mit dem <Code%> zurück; siehe
                          Abschnitt 'Tastatureingabe'. Beispiel: CHR$(65) ==> "A"
```

```
Priorität der Operatoren
------------------------
höchste ==>    (   )                          |                  NOT
~~~~~~~       AND                             |                  ^
              OR                              |                  *   /   MOD
              =   <   >   <>   <=   >=        |  niedrigste ==>  +   -
                                              |   ~~~~~~~~~~
```

*** zum Inhalt ***

*** Textausgabe auf Bildschirm, Farben**

****

```
- CLS - löscht den Bildschirm (färbt ihn schwarz bzw. mit der über COLOR vorge-
    gebenen Hintergrundfarbe ein; bei PowerBasic unter V3.5 immer schwarz!).
- LOCATE [<Zeile>] [, <Spalte>] - setzt den Cursor auf die angegebene Stelle
    (bei Textbildschirm (Screen 0): 25 Zeilen a' 80 Spalten bzw. die durch
    WIDTH definierte Aufteilung, s.u.). Bei Grafikbildschirmen (siehe
```

SCREEN 1...13) hängt die max. Anzahl Spalten/Zeilen vom Grafikmodus ab.

- LOCATE [<Zeile>], [<Spalte>], 1, 3, 5
  Cursor blinkend an gewählter Zeile u. Spalte setzen. Der Cursor erstreckt sich beim angegegebenen Beispiel über die 3. bis 5. Pixelzeile; siehe INTEXT.BAS.
- LOCATE , , 0 – Cursor wieder deaktivieren (erscheint nach späteren LOCATE-Befehlen nicht mehr von selbst)
- PRINT "text" [;|,] – gibt Text an der Cursorposition aus und setzt d.Cursor auf den Anfang der nächsten Zeile (Ausnahme: bei ";" am Ende bleibt der Cursor hinter dem Text, bei "," wird der Cursor auf die nächste 14-ner Spalte gesetzt {9/26}. Anführungszeichen " lassen sich über CHR$(34) einfügen. Schreibfaule können auch '?' statt 'PRINT' eingeben.
- PRINT "Zahl"; anna% – Gibt das Wort 'Zahl' und anschließend die in anna% gespeicherte Zahl aus (bei ',' statt ';' wird nach "Zahl" einem Tabulator eingefügt (Tabulator heißt Sprung auf Spalte n*14)
- PRINT ohne Zusatz – gibt Leerzeile aus
- PRINT TAB(18); "Hallo" – Cursor auf Spalte 18 setzen, dann "Hallo" ausgeben; die Zeichen zwischen der alten Cursorposition und Spalte 18 werden mit Leerzeichen überschrieben, d.h. gelöscht {9/27}.
- PRINT SPC(10) – 10 Spaces (Blanks) ausgeben, z.B. zum Löschen von Bildschirmausgaben. Mit SPC lassen sich bei SCREEN 0 höchstens 79 Blanks ausgeben (letztes Zeichen der Zeile nicht überschreibbar: SPC(80) funktioniert seltsamerweise nicht!). SPC ist nur in Print-Befehlen, nicht in Wertzuweisungen und Ausdrücken möglich (dort kann man nur SPACE$ verwenden).
- PRINT STRING$ (<Anz>, <text$>) – gibt Anz-mal das 1. Zeichen von Text$ aus z.B.: PRINT STRING (12, "_")
- PRINT USING <Maske$>; <Ausdruck> – formatierte Bildschirmausgabe mit Maske {3/68} {4/10f} {4/24} {9/30+43+141} {6/209} {11/308};
  Beispiel: PRINT USING "##.##"; 200/3 ==> 66.67 (mit Rundung, z.B. für Geldbeträge u.tabellarische Ausgaben)
- VIEW PRINT <AnfZeile> TO <EndZeile> – legt Ausgabefenster für Bildschirmausgabe fest; z.B.: VIEW PRINT 5 TO 24 'legt Ausgabefenster Zeile 5 bis 24 für die folgenden PRINT-Anweisungen fest. Die Ausgabe erfolgt dort rollierend, gut geeignet für die Anzeige von Tabellen, wenn die Tabellenüberschrift erhalten bleiben soll (siehe JOYTEST.BAS).
- COLOR [<Vordergr.farbe>] [, <Hintergr.farbe>] – Bildschirmfarbe für Textbildschirm (SCREEN 0) angeben; Vordergrundfarbe = Textfarbe {1/31} {1/61}
  Beispiele: COLOR 0,7 = schwarze Schrift auf hellgrauem Grund
  COLOR 14,1 = gelbe Schrift auf blauem Grund
  COLOR 15,0 = Schwarz/Weiß-Bildschirm wiederherstellen
  Der gesamte Bildschirm läßt sich durch anschließendes CLS mit der Hintergrundfarbe einfärben.

DOS-Farbcodes: (Farbcode+16 bei Vordergrundfarbe bewirkt blinkenden
============== Text; funktioniert nicht unter Win3.1/95)
(0...7 = dunkle Grundfarben, Addition von 8 ergibt jeweils die gleiche Farbe in hell {7/47})

0= schwarz	4= rot	8= grau	12= hellrot
1= blau	5= violett	9= hellblau	13= rosa
2= grün	6= braun	10= hellgrün	14= gelb
3= helltürkis	7= hellgrau	11= sehr helles Türkis	15= weiß

In Screen 0 sind als Hintergrundfarben nur die ersten 8 Farben darstellbar.

Die Farbcodes 8...15 werden wie die Farbcodes 0...7 dargestellt!.

- POS (0) - Systemvariable, liefert die aktuelle Spaltenposition des Cursors
- CSRLIN - Systemvariable, liefert die aktuelle Zeilenposition des Cursors
- WIDTH <Spaltenzahl>, <Zeilenzahl> - legt die Anzahl der Spalten und Zeilen fest (bei Textbildschirm SCREEN 0 und VGA-Monitor z.B. Spalten x Zeilen= 40 x 25, 40 x 43, 40 x 50, 80 x 25, 80 x 43 oder 80 x 50; in SCREEN 12 auch 60 Zeilen möglich!); bei EGA 80x25 oder 80x43, bei CGA 40x25 oder 80x25 Spalten x Zeilen möglich.
  Bei WIDTH 40,25 wird im DOS-Fenster von Win31/95 nur ein halbgroßes Fenster dargestellt.

//////////////////////////////// Für Profis /////////////////////////////////

- WIDTH "SCRN:", <Spaltenzahl> - Breite der Ausgabezeilen festlegen {11/464}.
  Bei Spaltenzahl=40 erscheint unter Win3.1/95 ein halb breiter Bildschirm.
- SCREEN (<Zeile>, <Spalte> [,1]) - Bildschirminhalt auslesen: Funktion, die den ASCII-Code des an der angegebenen Bildschirmposition angezeigten ASCII-Zeichens [bzw. dessen Farbwert] als INTEGER-Wert zurückliefert (muß vor einer erneuten Anzeige per PRINT mittels CHR$ wieder in ein Textzeichen rückgewandelt werden (siehe {11/400} und SCREENRD.BAS).
  Beispiel: Erste Bildschirmzeile auslesen und in t$ eintragen:

```
FOR i% = 1 TO 80: t$ = t$ + CHR$(SCREEN(1,i%)): NEXT i%
```

- WRITE <Variable1> [<Variable2, ...] - Selten verwendete Methode, Datensätze auf dem Bildschirm anzuzeigen; Darstellung wie im Abschnitt. 'Sequentielle Dateien' beschrieben (Strings in Anführungszeichen, Kommas zwischen den Variablen)

*** zum Inhalt ***

* Tastatureingabe

****

- INPUT ["",] <Variable> - liest Wert von Tastatur ein u.legt ihn in Variable ab ; ["",] unterdrückt das Fragezeichen. Kann die Variable Kommas enthalten, so ist LINE INPUT zu verwenden (s.u.).
- INPUT [;] "Gib Zahl ein" {;|,} Zahl1% - Kombinierte Bildschirm-Ausgabe und Tastatureingabe in die Variable 'Zahl1%' {1/23} ; das 1. Semikolon verhindert den Zeilenvorschub nach Abschluß der Anwender-Eingabe durch Enter. Wird das 2. Semikolon durch ein Komma ersetzt, so erscheint kein Fragezeichen.
- SLEEP - Warten bis beliebige Taste betätigt (Quick'n Dirty: Der 15 Zeichen umfassende Tastaturpuffer wird nicht gelöscht!)
- INKEY$ - liest ein Zeichen von der Tastatur; im Gegensatz zu INPUT wird nicht automatisch auf eine Eingabe gewartet. Beispiele:

```
- IF INKEY$ = CHR$(27)        'wenn Esc-Taste betätigt
- WHILE INKEY$ = "" :WEND     'warten bis eine beliebige Taste betätigt;
                              'kann durch x$ = INPUT$(1) (s.u.) oder
                              'Quick'n Dirty durch SLEEP ersetzt
                              'werden (s.o.)
- DO: taste$ = INKEY$         'Warteschleife bis beliebige Taste betätigt
    <Tastenbearbeitung>
  LOOP WHILE Taste$ = ""
- DO                          'Abfrage von Auswahltasten (Enter nicht er-
    Taste$ = INKEY$           'forderlich)
    SELECT CASE Taste$        'Gehe zur aktuellen Taste (String) {6/115}
      CASE "1": CALL Sub1     'Taste "1" betätigt
      CASE "2": CALL Sub2     'Taste "2" betätigt
      CASE "a" TO "m": CALL Suba  'eine der Tasten "a"..."m" betätigt
```

```
      END SELECT
    LOOP WHILE Taste$ <> CHR$(27)    'Ende mit Esc-Taste
  - Tastencodes für die Sondertasten (siehe {8/20} und TASTCODE.BAS; die
    "Buchstaben" sind als Großbuchstaben anzugeben):

      +-- Taste --+-- Code ---+---- Taste ----+------ Code ----------------+
      | Enter     = CHR$(13)   | Cursor hoch    = CHR$(0) + "H"             |
      | Leertaste = CHR$(32)   | Cursor tief    = CHR$(0) + "P"             |
      | Backspace = CHR$(8)    | Cursor links   = CHR$(0) + "K"             |
      | Esc    = CHR$(27)      | Cursor rechts = CHR$(0) + "M"              |
      | Einf = CHR$(0) + "R"   | Bild hoch      = CHR$(0) + "I"             |
      | Entf = CHR$(0) + "S"   | Bild tief      = CHR$(0) + "Q"             |
      | Pos1 = CHR$(0) + "G"   | F1....F10      = CHR$(0) + CHR$(59).....(68)|
      | Ende = CHR$(0) + "O"   | F11...F12      = CHR$(0) + CHR$(133)...(134)|
      +------------------------+-------------------------------------------+
      Anwendungsbeispiele:
        IF INKEY$ = CHR$(0) + "H" THEN       'wenn Cursor hoch betätigt
        IF INKEY$ = CHR$(0) + CHR$(60) THEN  'wenn F2-Taste betätigt
  - WHILE INKEY <> "":WEND     'Tastaturpuffer leeren

////////////////////////////////  Für Profis  /////////////////////////////////
- Ereignisgesteuerte Tastenbearbeitung (Tasteninterrupt):
    - ON KEY (<Tastennr. 1...31>) GOSUB <Name der Subroutine> - Ereignisgesteu-
        ertes Aufrufen einer Subroutine, wenn eine Taste betätigt wird  {4/38}
        {11/359}
          - Die Subroutine muß als 'lokale Subroutine' im Hauptprogramm defi-
            niert sein (siehe entsprechenden Abschnitt)
          - Tastennr. = 1...10|30|31   ==> Funktionstasten F1...F10|F11|F12;
               F1 z.B. für Hilfefunktion verwendbar {11/362}
          - Tastennr. = 11|12|13|14    ==> Cursortasten  Hoch|Links|Rechts|Tief
          - Tastennr. = 15...25        ==> benutzerdefinierte Tasten (siehe Hilfe
               zu KEY und {11/461f}). Diese Tasten werden wie folgt definiert:
                 - KEY <Tastennr.>, CHR$(<Tastenstatus%>) + CHR$(ScanCode%)
                     - Der Tastenstatus kennzeichnet Zusatztastenbetätigungen:
                         0    = keine Zusatztaste gedrückt | 8 = Alt
                         1,2,3= Shift zusätzlich gedrückt  | 32= NumLock aktiv
                         4    = Strg  zusätzlich gedrückt  | 64= ShiftLock aktiv
                - Der Scancode geht aus der QBasic-Hilfe hervor, erreichbar
                  über <Inhalt> <Kurzübersicht | Tastatur-Abfragecodes>
    - Die Steuerung der Ereignisverfolgung erfolgt über  {11/463}
        KEY (<Tastennr.>) {ON | OFF | STOP}
           - Tastennr.=0 ==> Die Steuerung der Ereigniserfolgung geschieht für
                             alle Tasten gemeinsam
           - ON          ==> Ereignisvefolgung aktivieren
           - OFF         ==> Ereignisverfolgung deaktivieren
           - STOP        ==> Ereignisverfolgung aktiviert, Ausführung erfolgt
                             jedoch erst nach KEY ... ON
    - Beispiel 1: KEY(1) ON      'Ereignisverfolgung für F1-Taste aktivieren
                  ON KEY(1) GOSUB Hilfe  'Subroutine "Hilfe" aufruf. bei F1
                  [Key(1) OFF]  'Ereignisverfolgung deaktivieren
                  [Key(1) STOP] 'Überwachung der Taste unterbrechen, jedoch Tas-
                                'tenbetätigungen weiter registrieren und nach
                                'KEY(1) ON zur Wirkung kommen lassen.
    - Beispiel 2: Key 15, CHR$(0) + CHR$ {(51)| {1)} 'Komma- | Esc-Taste als
                                'benutzerdefinierte Taste 15 definieren
```

```
ON KEY(15) GOSUB TuNix 'TuNix muß s.i.Hauptprogramm befinden
KEY(15) ON
...
TuNix: PRINT "Ich Tu Nix": RETURN       'siehe MATHEFIX.BAS
```

- ON KEY(<Zahl 1...31>) GOTO <Sprungmarke$> - Ereignisgesteuerte Tastenbearbeitung mit Direktsprung statt Aufruf einer Subroutine; ansonsten wie oben (wenig gebräuchlich {11/360}).

- Funktionstasten mit Zeichenketten belegen (für Menüs usw. {11/460})
  - KEY <Tastennr.>, <Zeichenkette$> - Funktionstasten (Tastennr.: siehe oben) mit einer Zeichenkette von max 15 Zeichen belegen
  - KEY ON - Anzeige der (max. 6) Funktionstastenbelegungen in der unteren Bildschirmzeile aktivieren
  - KEY OFF - Anzeige der (max. 6) Funktionstastenbelegungen in der unteren Bildschirmzeile deaktivieren
  - KEY LIST - Komplette Liste aller Funktionstastenbelegungen anzeigen
- <Stringvariable$> = INPUT$(<n%>) - Spezialfunktion: Warten bis n Zeichen über die Tastatur eingegeben wurden (ohne Echo!); diese Zeichen werden in die Stringvariable eingetragen {9/72} {7/60}. Hierfür gibt es nur wenige praktische Nutzanwendungen; höchstens vielleicht die folgende:

  ```
  x$=INPUT$(1)   'warten bis beliebige Taste betätigt
  ```

  Der Cursor kann über speziellen LOCATE-Befehl zur Anzeige gebracht werden {7/31}.
- LINE INPUT [;] ["Eingabeaufforderung";] <Stringvariable>
  Einlesen einer kompletten Textzeile inklusive Kommas, welche sonst als Trennzeichen zwischen Eingabewerten dienen. Ein Fragezeichen wird nicht ausgegeben; [;] bewirkt, daß der Cursor in der Eingabezeile stehenbleibt {6/266}

** zum Inhalt ***

* **Auf Grafikbildschirm zeichnen** (geht nicht im Textmodus Screen 0)

****

- Grafik-Bildschirmkoordinaten und ihre Verschiebung/ Skalierung:
  - Alle Koordinaten und Längenangaben werden normalerweise in Anzahl Pixeln angeben (x,y = 0...max-1).

```
                                            +---------------------+ -->x
Beispiel: VGA-Bildschirmkoordinaten (x,y):  |(0,0)   VGA   (639,0)| |
                                            |(0,479)     (639,479)| v
                                            +---------------------+ y
```

  - Über STEP lassen sich bei vielen Grafikbefehlen relative Koordinaten aktivieren - bezogen auf die momentane Position des Grafikcursors.
  - Eine Skalierung der Koordinaten ist mit dem WINDOW-Befehl möglich (multiplikative Beeinflussung des Maßstabs; siehe unten).
  - Zum Positionieren des Textcursors dient auch bei den Grafikbildschirmen (Grafikmodus >0) der - nicht pixelorientierte - LOCATE-Befehl, zum Festlegen der Spalten/Zeilenzahl der WIDTH-Befehl (siehe 'Textausgabe...').
- SCREEN <Grafikmodus> - Grafikbildschirm-Auflösung wählen {1/31+17} {11/170} (SCREEN 0 vorbesetzt). Die gebräuchlichsten Grafikmodi sind:

  SCREEN 0 = Textmodus, für alle Grafikkarten, läuft als einziger Bildschirmmodus auch problemlos im DOS-Teilfenster von Windows, 16 Farben, 8 Bildschirmseiten (0-7)

  SCREEN 1 = CGA/EGA/VGA-Karte, 320*200 Grafik, 30*25 Text, 4 aus 16 Farben, [2 Bits pro Pixel in 1 Ebene für GET/PUT], 1 Bildschirmseite (0)

  SCREEN 2 = CGA/MCGA/EGA/VGA-Karte, 640*200 Grafik, 80*25 Text, 2 aus 16 Farben, [1 Bit pro Pixel in 1 Ebene für GET/PUT], 1 Bildschirmseite (0)

SCREEN 7 = EGA/VGA-Karte, 320*200 Grafik, 40*25 Text, 16 Farben, 8 Bildschirmseiten (0-7). Ruckelfreue Animationen auch auf langsamen Rechnern möglich {1/75}, [4 Bits pro Pixel in 4 Ebenen für GET/PUT]

SCREEN 9 = EGA/VGA-Karte, 640*350 Grafik, 80*15 Text, bis 16 Farben, [4 Bits pro Pixel (bei 16 Farben) in 4 Ebenen für GET/PUT] 2 Bildschirmseiten (0-1)

SCREEN 11= VGA-Karte, 640*480 Grafik, 80*25|30|50|60 Text (Voreinstellung: 80*30), 2 aus 256 Farben, gut geeignet für s/w-Grafiken [1 Bit pro Pixel in 1 Ebenen für GET/PUT], 1 Bildschirmseite

SCREEN 12= VGA-Karte, 640 x 480 Grafik, 80*30|50|60 Text (Voreinstellung: 80*80*30), 16 aus 256 Farben, eine Bildschirmseite, [4 Bits pro Pixel in 4 Ebenen für GET/PUT], 1 Bildschirmseite

SCREEN 13= VGA-oder MCGA-Karte, 320 x 200 Grafik, 40*25 Text, 256 Farben, eine Bildschirmseite, [8 Bits pro Pixel in 1 Ebene für GET/PUT], von PowerBasic nicht unterstützt. 1 Bildschirmseite.

- SCREEN <Grafikmodus>, , <Ausgabeseite>, <Anzeigeseite> - Bildschirm-Ausgabeseite und Anzeigeseite umschalten. Die Anzahl der zur Verfügung stehenden Bildschirmseiten ist in der QBasic-Onlinehilfe unter <SREEN | Bildschirm-modi> abfragbar; sie hängt vom Grafikmodus ab und kann bis zu 8 betragen. Die Verwendung mehrer Seiten kann Animationen ruckfrei machen{1/75}{11/172}.
- PCOPY <Quellseite%>, <Zielseite%> - Inhalt einer Bildschirmseite in eine andere kopieren {11/464}
- COLOR - Farbe verwenden; Syntax hängt vom verwendeten SCREEN ab.
  Beispiele: SCREEN 1 : COLOR <Hintergrundfarbe>, <Farbpalette>
  {11/185} SCREEN 7|8|9 : COLOR <Zeichenfarbe>, <Hintergrundfarbe>
  SCREEN 12|13 : COLOR <Zeichenfarbe>
  Bei Grafikbildschirm (z.B. SCREEN 12) wird der Hintergrund durch CLS: PAINT (x,y), <Farbcode> eingefärbt (x,y beliebig).
- LINE - Linie oder Viereck zeichnen {1/33} {11/175}; gestrichelte Linie durch Anhängen von [,&H<16-Bit-Hexa-Zahl>] möglich (endlos wiederholtes Pixelmuster: 0|1= Linienpixel nicht vorhanden|vorhanden {11/191})
  - LINE [(x1,y1)]-(x2,y2) [,<Farbcode>] - farbige Linie von P1 nach P2
  - LINE [(x1,y1)]-(x2,y2) ,[<Farbcode>], B [F] - Viereck (Box) mit der Diagonalen P1-P2 zeichnen, [F=mit Farbe ausgefüllt]
  - LINE [STEP (x1,y1)] - STEP (x2,y2)... - dito mit relativen Koordinatenangaben bezogen auf die momentane Cursorposition {7/42}

  Bei weggelassenem Anfangspunkt P1 (x1,y1) wird die momentane Position des Grafikcursors als Anfangspunkt verwendet.
- CIRCLE - Kreis zeichnen {1/33} {7/42} {11/182}, auch für Ellipsen u.Kreisbögen
  - CIRCLE [STEP] (x,y), <Radius> [,<Farbcode für Kreislinie>] - Kreis zeichnen. [STEP] definiert x und y als relative Koordinaten, bezogen auf die momentane Cursorposition.
    Der Kreis läßt sich mit Farbe füllen über PAINT (x,y), <Farbcode>
  - CIRCLE (x,y), <Radius>, [Farbcode],,, <Faktor> - Ellipse mit Stauchungs-Faktor Höhe/Breite zeichnen {11/182+198}; die Ellipse paßt immer in den Kreis mit dem angegebenen Radius hinein (Faktor < 1 ==> Breite = Radius; Faktor > 1 ==> Höhe = Radius)
  - CIRCLE (x,y), <Radius>, [Farbcode], <Anfangswinkel>, <Endwinkel> [, Faktor>] - Kreisbogen [Ellipsenbogen] zwischen Anfangs- und Endwinkel zeichnen (Winkelangaben im Bogenmaß (d.h. in Radian: 3,14 Radian= pi#=180°), oben = 0°, wird im Uhrzeigersinn gezeichnet {11/183}).
    Beispiel: 3/4-Ellipsenbogen um den Punkt (200,100) mit dem Radius 50 Pixel von 180° nach 90° ziehen; Breite= 50 Pixel, Höhe= 50*0.4=20 Pixel:

CIRCLE(200, 10), 50, 3.14, 3.14/2, 0.4

- PAINT [STEP] (x,y) - vorher mit LINE und/oder CIRCLE begrenzte Fläche mit der Randfarbe einfärben (STEP macht die Koordinaten relativ {11/189})
- PAINT [STEP] (x,y), <Füllfarbe%>, <Randfarbe%> - Fläche einfärben bis die Randlinie mit der angegebenen Randfarbe erscheint
- PAINT [STEP] (x,y), <Muster$>, <Randfarbe> - Fläche mit Muster$ ausfüllen bis die Randlinie mit der angegebenen Farbe scheint. Muster$ wird im Binärcode interpretiert und 1-Positionen mit der aktuellen Zeichenfarbe in horizontal wiederholten Reihen eingefärbt (siehe {11/190} und MUSTER.BAS).
- PSET [STEP] (x,y)[, <Farbcode>] - einen Bildpunkt (Bildschirmpixel) malen {7/40}. STEP macht die x/y-Koordinaten relativ.
- PRESET [STEP] (x,y) - Bildpunkt löschen (mit Hintergrundfarbe übermalen)
- DRAW <Befehls-String$> - Polygonzug zeichnen, verketteter Befehl zum Zeichnen aneinanderhängender Linien mit einem gedachten Zeichenstift, der gleichzeitig dem Grafikcursor entspricht (ähnlich dem Sound-PLAY-Befehl).
  Teilelemente des Befehls-String$: {7/42} {2/36} {11/218}
  - <Richtungsbuchstabe$> [<n%>] - bewegt den Zeichenstift um 1 [oder n%] Pixel in die durch den Richtungsbuchstaben definierte Richtung und zeichnet eine entsprechende Linie.

```
                         H  U  E
                          \ | /
Richtungsbuchstaben:     L--+--R
                          / | \
                         G  D  F
```

  - M [+|-] x, [+|-] y - bewegt den Zeichenstift auf [um] die Koordinaten x, y und zeichnet eine entsprechende Linie. + bzw. - bewirkt eine Bewegung um relative Koordinaten.
  - B - Präfix: Zeichenstift heben und ohne zu zeichnen bewegen.
  - N - Präfix: Nach dem Zeichnen Zeichenstift wieder auf Ausgangsposition setzen
  - C <n%> - Zeichenfarbe setzen
  - A <n%> - Zeichenstift um n%*90° entgegen dem Uhrzeigersinn drehen (n%=1, 2 oder 3) bzw.das 'Zeichenblatt' unter dem Zeichenstift um 90° im Uhrzeigersinn drehen. Die Richtungsbuchstaben ändern ihre Wirkungsrichtung entsprechend.
  - TA <n%> - Zeichenstift um n° drehen (n%=-360..+360). Die Richtungsbuchstaben ändern ihre Richtung entsprechend. {11/221}
  - An%|Bn% - Objekt um n% Grad rotieren | Zeichenfarbe setzen
  - P n1%,n2%- Füll- und Randfarbe eines Objektes setzten
  - S n% - Längeneinheit für Zeichenstiftbewegung setzen (4 entspricht 1 Pixel)
  - Beispiel: Dreick zeichnen durch Verbindung der 3 Punkte (200,50), (250,50) und (250,20) mit roten Linien: {11/221}

```
SCREEN 12: DRAW "C4 BM200,50 R50 U30 M200,50"
```

////////////////////////////////// Für Profis //////////////////////////////////

- PALETTE <Farbkennziffer%>, <Farbcode&> - Einer bestimmten Kennziffer einen neuen Farbcode dynamisch zuordnen (funktioniert u.U. nur im Screen 1). Der Zahlenbereich des Farbcodes& hängt von der Grafikkarte ab {11/186}.
- PALETTE USING <Feld[(<Startindex>)]> - Farbpalette einem Feld zuweisen, Feld muß vorher mit Farbcodes& gefüllt werden {11/186}
- POINT (x, y) - Funktion, die den Farbcode des Bildschirmpunktes (x, y) zurückliefert
- VIEW [SCREEN] (x1,y1) - (x2,y2) - Bildschirmausschnitt als aktuelle Grafik-

fläche definieren, die mit CLS selektiv gelöscht werden kann.
  - Dieser Befehl ist gut geeignet zum Zeichnen von kleinen animierten Grafiksymbolen (Sprites), die mit GET/PUT angezeigt und abgespeichert werden sollen; siehe {11/197..201} und GETPUT1.BAS.
  - Bei VIEW ohne SCREEN beziehen sich alle in den nachfolgenden Befehlen verwendeten Koordinaten auf die linke obere Ecke des Bildschirmausschnitts (d.h. der Punkt (0,0) ist identisch mit (x1, y1).
  - Bei Angabe von SCREEN beziehen sich beziehen sich alle Koordinaten nach wie vor auf die linke obere Ecke des Gesamtbildschirms.
- VIEW - (ohne Parameter): Die obige Bildschirmausschnitt-Definition wieder aufheben {11/198}
- Mit GET/PUT läßt sich Grafik in einem RAM-Bildfeld speichern/ Bildfelddaten als Grafik anzeigen. Hierdurch lassen sich Bildelemente bequem in RAM-Feldern speichern und schnell auf den Bildschirm ausgeben - ohne langwieriges Neuzeichnen (siehe {11/202+205} und GETPUT1.BAS):
  - DIM <Bitfeldname>% (<laenge>) - INTEGER-Bildfeld zur Ablage eines Bildelements deklarieren. Die erforderliche <laenge> des Feldes hängt vom verwendeten SCREEN-Grafikmodus und der Hoehe/ Breite des Bildschirmausschnitts wie folgt ab (siehe {11/205} und QBasic-Online-Hilfe unter <PUT | Bilddatenfelder und Kompatibilität>:

```
+------------------------------------------------------------------+
| laenge& des für eine Grafik-Get/PUT-Operation benötigten Feldes = |
|    (4 + Hoehe*Ebenen*INT((Breite*BitsProPixel/Ebenen + 7)/8))\2 + 1 |
+------------------------------------------------------------------+
```

    mit - Hoehe = Höhe des Bildschirmausschnitts y2-y1+1 (siehe GET)
    - Breite = Breite des Bildschirmausschnitts x2-x1+1 (siehe GET)
    - Ebenen = Anzahl der Farbebenen (abhängig vom Grafikmodus, siehe oben bei den SCREENS 1...13)
    - BitsProPixel = Speicherbedarf je Bildschirmpixel (abhängig vom Grafikmodus (siehe oben bei den SCREENS 1...13)
  - GET [STEP] (x1,y1) - [STEP] (x2,y2), <Bildfeldname> - Rechteckigen Ausschnitt des Anzeigebildschirms in Bildfeld einlesen. (x1,y1)= obere linke, (x2,y2)= rechte untere Ecke. STEP macht die Koordinaten relativ. Die erforderliche Länge des Bildfeldes ergibt sich aus der Formel im obenstehenden Kasten.
  - PUT [STEP] (x1,y1), <Bildfeldname>% - Grafikinformation aus dem Bildfeld auf den Bildschirm an der durch (x1,y1) gekennzeichneten Stelle zur Anzeige bringen. (x1,y1)= Koordinaten der linken oberen Ecke. Die alte auf dem Bildschirm angezeigte Grafikinformation wird vorher gelöscht. STEP macht die Koordinaten relativ.
    Die erforderliche Länge des Bildfeldes ergibt sich aus der Formel im obenstehenden Kasten.
  - PUT (x1,y1), <Bildfeldname>%, {PSET|PRESET|AND|OR|XOR} - Entspricht dem obigen PUT-Befehl, jedoch wird die alte Anzeigeinformation nicht gelöscht, sondern wie folgt mit der Grafikinformation des Bitfeldes verknüpft (siehe {11/207} und GETPUT2.BAS):
    - PSET = löscht vorhandenen Bildausschnitt, fügt neues Bild ein
    - PRESET = löscht vorhandenen Bildausschnitt, invertiert neues Bild
    - AND = mischt neues mit vorhandenem Bild: Nur Bildpunkte, die im alten und neuem Bild gezeichnet sind, erscheinen auf dem Bildschirm.
    - OR = überlagert vorhandenes mit neuem Bild: Alle gezeichneten Bildelemente des alten und des neuen Bildes werden dargestellt (z.B. für transparente Sprites!!)
    - XOR = überlagert vorhandenes mit neuem Bild: Wie OR, jedoch blei-

ben diejenigen Bildpunkte dunkel, die im alten und im neuen Bild Zeichenelemente erhalten.

- WINDOW [SCREEN] (x1!,y1!) - (x2!,y2!) - Skalierung der x/y-Koordinaten: Die nachfolgenden Grafikbefehle verwenden nicht Pixelkoordinaten, sondern 'virtuelle' Koordinaten. Die durch virtuelle Koordinaten angegebenen Punkte (x1!,y1!) und (x2!,y2!) entsprechen den Eckpunkten des Gesamtbildschirms bzw. des mit einem vorangegangenen VIEW-Befehl definierten Bildschirm-Ausschnitts (siehe oben). Siehe auch {11/213} und SINUS.BAS
  - Bei weggelassenem SCREEN kehrt sich die Wirkungsrichtung der x/y-Koordinaten um (y_unten= 0, y_oben= max; wie in der Mathematik).
  - Bei Verwendung von SCREEN hat die y-Achse die gleiche Wirkungsrichtung wie bei normalen Pixelkoordinaten. Beispiel für VGA-Bildschirm:

```
          +---------------------+         +---------------------+
 ohne     |            (x2!,y2!)|   mit   |(x1!,y1!)            |
 SCREEN:  |(x1!,y1!)            | SCREEN: |            (x2!,y2!)|
          +---------------------+         +---------------------+
```

    Zur Umrechnung von Pixel- in virtuelle Koordinaten und umgekehrt stehen die Befehle POINT und PMAP zur Verfügung:

- POINT <Modus%> - liefert als Zielwert die aktuelle Position des Grafikcursors in Pixelkoordinaten bzw. virtuellen Koordinaten (siehe WINDOW-Befehl) gemäß der folgenden Tabelle zurück {11/216}:

Modus%	Startwert	Zielwert
0	x-Koordinate virtuell *)	x-Koordinate, Pixel
1	y-Koordinate virtuell *)	y-Koordinate, Pixel
2	x-Koordinate, Pixel	x-Koordinate virtuell *)
3	y-Koordinate, Pixel	y-Koordinate virtuell *)

  *)= virtuelle Koordinaten= mittels WINDOW-Befehl skalierte Koordinaten.
  Der Startwert ist für den POINT-Befehl ohne Bedeutung.

-PMAP (<Startwert>, <Modus%>) - Rechnet den Startwert entsprechend obiger Tabelle um und liefert den Zielwert als Ergebnis zurück {11/216}.

*** zum Inhalt ***

*** Sound aus PC-Speaker ausgeben**

****

- BEEP - einen Piepston erzeugen
- SOUND <FrequenzInHerz%>, <DauerInSystemtakten%> - einfache Art der Soundausgabe: Es wird ein Ton mit der angegebenen Frequenz der Dauer in Systemtakten à 56 ms ausgegeben; siehe Sirene in {2/35}, {6/134}, {11/224,}, KLAVIER.BAS und MUSIK.BAS.

```
  Beispiele: - SOUND 2000, 6           '2000Hz-Ton 6*55ms=330ms lang spielen
             - DO: SOUND 192, 0.5: SOUND 188, 0.5 : LOOP    'Motorengeräusch
```

- PLAY <Befehls-String$> - Komfortable Ausgabe von Musik und Soundeffekten über den PC-Speaker (siehe {7/16}, {11/222} {1/27} und KLAVIER.BAS.
  Teilelemente des Befehls-String$:
  ------------------------ Noten spielen, Oktave festlegen -----------------
  - M{F|B} - alle folgenden Noten im Vordergrund|Hintergrund abspielen (Foreground|Background). 'Vordergrund' bedeutet, daß mit der Abarbeitung der Folgebefehle solange gewartet wird bis der PLAY-Befehl komplett ausgegeben worden ist. 'Hintergrund' bedeutet, daß während des Spielens das Programm fortgesetzt wird. Vorbesetzung= MF
  - {A|B|...|G|} - Note a, h, c, d, e, f oder g der Tonleiter in der aktuellen Oktave spielen
  - O<n%> - aktuelle Oktave für die folgenden Noten festlegen (n=0...6)

```
   - N<n%> - einen Ton aus dem gesamten 7-Oktav-Bereich spielen (n=0...84,
             0=Pause)
   - <     - eine Oktave erhöhen, gilt für alle nachfolgenden Töne
   - >     - eine Oktave erniedrigen, gilt für alle nachfolgenden Töne
  ------------------------ Tonlänge, Tempo, Pausen -------------------------
   - L<q%> - Länge der nachfolgenden Töne festlegen (q=1-64; Tonlänge = 1/q;
             1 ist eine ganze Note; Vorbesetzung: q = 4 ==> 1/4 Note)
   - P<q%> - Pausendauer zwischen den nachfolgenden Töne festlegen (q=1-64;
             Pausendauer = 1/q; Vorbesetzung: q = 4 ==> 1/4 Note)
   - T<q%> - Tempo der nachfolgenden Noten in Viertelnoten/min festlegen;
             (n=32-255); Vorbesetzung: n= 128
  -------------------------- Suffixe für Einzelnoten -----------------------
   - {+|#} - Suffix: Die vorangehende Note um einen Halbtonschritt erhöhen
   -  -    - Suffix: Die vorangehende Note um 1 Halbtonschritt erniedrigen
   -  .    - Suffix: Die vorangehende Note 1,5 mal so lang spielen
  --------------------------Staccato und Legato ----------------------------
   - MS    - alle nachfolgenden Noten in Staccato spielen (kurz und abge-
             hackt, nicht mit dem nächsten Ton verbunden)
   - ML    - alle nachfolgenden Noten in Legato spielen (lang und getragen,
             mit der nächsten Note verbunden)
   - MN    - alle nachfolgenden Noten wieder normal spielen (nicht Staccato
             oder Legato)
  ------------------------------ Beispiel ----------------------------------
   - PLAY          "MB ML T160 O1 L2 gdec P2 fedc"          'Big-Ben-Schlag
          im Hinter- |  |   |  |   |   |   |   |
           grund   --+  |   |  |   |   |   |   +-- letzte 4 Noten
          Legato -------+   |  |   |   |   +------ 1/2 Notenlänge Pause
          Tempo 160 --------+  |   |   +---------- erste 4 Noten
          1.Oktave ------------+   +-------------- Notenlänge: 1/2 Note
```

//////////////////////////////// Für Profis ////////////////////////////////

- ON PLAY (<Notenanzahl%>) GOSUB <Marke$> - Ereignisgesteuertes Anspringen der lokalen Subroutine <Marke$>, wenn der PLAY-Notenpuffer weniger noch ungespielte Noten als die angegebene <Notenanzahl%> enthält (z.B. =2). Die Subroutine enthält normalerweise einen Play-Befehl mit 'Notennachschub' für lange Hintergrundmusiken {11/370}
- PLAY {ON|OFF|STOP} - Ereignisverfolgung für Notenpufferauswertung aktivieren | deaktivieren | unterbrechen mit Speicherung
- PLAY(0) - liefert die Anzahl der gerade im PLAY-Notenpuffer stehenden noch ungespielten Noten zurück {11/370}

*** zum Inhalt ***

*** Joystickabfrage**

****

Siehe JOYTEST.BAS. Achtung: Joystick-Abfragewerte bei Sound-Ausgabe auf den PC-Speaker verfälscht! Abfrage von Joystick B: Siehe Online-Hilfe und {11/373}

- STICK(0) | (1) | (3) - X|Y-Achse| Schubregler abfragen, Wert 255 ... 0
- STRIG(1) | (5) - Wert -1 = Feuerknopf A|B betätigt (bei PowerBasic vorher mit STRIG ON Ereignisverfolgung aktivieren)
- ON STRIG (<Knopfnr%>) GOSUB <Marke$> - Ereignisgesteuertes Anspringen der lokalen Subroutine <Marke$> bei Drücken eines Joystick-Knopfes {11/373}
- STRIG {ON|OFF|STOP} - Ereignisverfolgung für Joystick-Knöpfe aktivieren|deaktivieren|unterbrechen mit Speicherung

*** zum Inhalt ***

* **Zeiten erzeugen und Datum/ Uhrzeit bearbeiten**

****

- SLEEP [<n%>] - Wartezeit n sec einlegen (nur ganze Sekunden); Die Wartezeit wird bei Betätigung einer beliebigen Taste vorzeitig abgebrochen. Beispiel: SLEEP 2 '2sec warten; SLEEP ist bei PowerBasic erst ab V3.5 vorhanden (bei älteren Versionen durch DELAY ersetzen). Bei SLEEP mit Parameter '0' oder ohne Parameter wird bis zur nächsten Tastenbetätigung gewartet, Tastaturpuffer jedoch nicht geleert {1/13}.
- TIMER - Systemuhr, zeigt die seit Mitternacht vergangenen sec mit einer Auflösung von 18,2 Inkrementen pro sec., d.h. von ca 56ms=0,056 s. Beispiel zur Erzeugung einer feinaufgelösten Wartezeit von 0,5s:
  ```
  starttime! = TIMER     'seit Mitternacht abgelaufene Zeit in s
  DO: LOOP UNTIL TIMER > starttime! + .5
  ```
  Der Timer liefert Gleitpunktwerte vom Typ SINGLE zwischen 0 und 86400 (entspricht den 24 Stunden von 00:00:00h ... 23:59:59h). Bei der Realisierung von Stoppuhren und Countdown-Timern ist der Rücksprung vom Maximalwert auf 0 um Mitternacht zu berücksichtigen.
- MTIMER - Nur bei PowerBasic vorhanden: Mikrotimer mit einer Auflösung von 1µs
- DATE$ - Datum ausgeben als String im Format MM-TT-JJJJ, z.B. "04-29-1999" (Umwandlung in deutsches Format: Siehe DAT-ZEIT.BAS). Systemdatum änderbar durch DATE$ = <Datum-String$>
- TIME$ - Uhrzeit ausgeben als String im Format HH:MM:SS, z.B. "18:58:12". Systemzeit änderbar durch TIME$ = <Uhrzeit-String$>
- ON TIMER (<AnzahlSekunden&>) GOSUB <Marke$> - Ereignisgesteuert (abhängig vom Timerinhalt) wird alle <AnzahlSekunden%> die lokale Subroutine <Marke$> angesprungen; AnzahlSekunden% kann einen ganzzahligen Wert zwischen 0 und 86399 annehmen (entspricht den 24 Stunden von 00:00:00h ... 23:59:59h) {11/366}.
- TIMER {ON|OFF|STOP} - Ereignisverfolgung für Timer aktivieren|deaktivieren |unterbrechen mit Speicherung.

*** zum Inhalt ***

* **Zufallszahlen erzeugen** {9/85} {2/5}

****

- RANDOMIZE TIMER - Zufallsgenerator auf Systemuhr-abhängigen, d.h. immer anderen Startwert setzen {1/45} {1/69}
- RND - liefert eine Zufallszahl vom Typ SINGLE zwischen 0 und 0.9999999; {1/45+65+69}
- Beispiele:
  ```
  RANDOMIZE TIMER          'ganzzahlige Zufallszahl z% erzeugen ...
  z% = INT(RND * 6) + 1    '... zwischen 1 und 6  oder ...
  z% = INT(RND * 90) + 10  '... zwischen 10 und 99 oder ..
  z% = INT(RND * (max%-min%+1))+min%  '..zwischen min und max (inkl.)
  ```
- Erzeugen von Zufallszahlen ohne Doubletten: Siehe RANDOMNO.BAS

*** zum Inhalt ***

* **Allgemeines zu Subroutinen und Funktionen (Parameter, Lokal-/Globalvariable)**

****

Das folgende gilt nicht für lokale SUBs und FUNCTIONs.

- Subroutinen und Funktionen werden von QBasic in eigenen Editierfenstern eingetragen/editiert. Dies macht ein QBasic-Programm äußerst übersichtlich, und eine SUB/FUNCTION läßt sich schnell und bequem auffinden. So einen Eintrag nennt man 'SUB/FUNCTION-Definition'. Die SUB/FUNCTION-Fenster sind über <Ansicht | SUBs...> oder die F2-Taste zugreifbar.
- Neue Subroutinen und Funktionen lassen sich über <Bearbeiten | Neue SUB...> anlegen.

- Eine Subroutine|Funktion muß im aufrufenden Hauptprogramm deklariert werden. Die Deklaration wird vom QBasic-Editor automatisch wie folgt ganz am Anfang des aufrufenden Hauptprogramms eingefügt:
  DECLARE SUB|FUNCTION <Name der Subroutine> ([<Formalparameter 1>, ...])
  Änderungen in der Parameterliste durch den Programmentwickler müssen in dieser Deklaration händisch nachgeführt werden.
  Anmerkung zu PowerBasic: Die Deklaration von SUBs/FUNCTIONs im Hauptprogramm ist nur erforderlich, wenn sich die SUB/FUNCTION in einer anderen Datei befindet. Als Formalparameter sind die Variablentyp-Bezeichner statt der Parameternamen anzugeben, z.B. 'LONG' statt 'hugo&'.
- Eine SUB | FUNCTION kann mit 'EXIT {SUB|FUNCTION}' vorzeitig verlassen werden
- Startwert der lokalen Variablen: Alle lokalen Variablen werden bei jedem Aufruf der SUB/FUNCTION mit dem Startwert '0' (bzw "" bei Stringvariablen) vorbesetzt. Dies gilt nicht für globale Variable (mit SHARED deklariert) und wiedereintrittsfähige Variable (mit STATIC deklariert; siehe unten).
- Geltungsbereich der Variablen {9/100} {3/133} {11/122}:
  - Variablen und Felder des Hauptprogramms sind in der SUB/FUNCTION nur zugreifbar, wenn sie im Hauptprogramm als globale Variable deklariert sind (siehe unten unter 'Variante 1') oder wenn sie als Parameter übergeben werden.
  - Lokale Variable und Felder einer SUB/FUNCTION sind vom Hauptprogramm aus nur zugreifbar, wenn sie in der SUB/FUNCTION als globale Variable deklariert sind (siehe unten unter 'Variante 2'). Sollen sie auch in anderen SUB/FUNCTIONs verwendet werden, so sind sie dort ebenfalls global zu deklarieren.
- Geltungsbereich der Konstanten: Im Hauptprogramm per CONST deklarierte Konstanten gelten auch in allen SUBs und FUNCTIONs.
- Übergabe von Feldern an SUBs und FUNCTIONs: Felder können als Parameter an die SUB oder FUNCTION übergeben werden (mit leeren Klammern ()). Eine nochmalige Deklaration des Feldes in der SUB/FUNCTION ist nicht erforderlich. Siehe {9/98}, {6/223}, RANDOMNO.BAS und FLDPARAM.BAS. Beispiel:

```
DECLARE SUB Upro(feld())          'Deklaration der SUB, wird von QBASIC
                                  'automatisch im Hauptprogramm eingefügt
DIM feldx(3,4)                    'Aufruf der Subroutine Upro und Über-
CALL Upro(feldx())                'gabe eines zweidimensionalen Feldes
SUB Upro(feldy()): feldy(2,3)=47  'Definition der SUB mit feldy als For-
                                  'malparameter
```

- Übergabe von anwenderdefinierten Feldern (Verbundfeldern anwenderdefinierten Typs) an SUBs und FUNCTIONs mit 'AS ANY' {11/271}: Beispiel:

```
DECLARE SUB Upro(feldx() AS ANY)  'Deklaration v.QBasic automat.eingefügt
                                  'Man kann auch 'AS quiz' angeben
DIM feldx(13) AS quiz...
CALL Upro(feldx())
SUB Upro(feld() AS quiz)          'AS <Typname> muß mit angegeben werden
```

//////////////////////////////// Für Profis ////////////////////////////////////

- Bei mit STATIC deklarierten lokalen Variablen und Feldern bleibt der Wert zwischen zwei Aufrufen der SUB/FUNCTION erhalten {9/98} {3/130}:
  - STATIC <Variablenname> [AS <Typ>], ...   'für Variablen
  - STATIC <Feldname> ()   'für Felder
    DIM <Feldname> (<Anzahl Feldelemente%>)
- Explizite Deklaration von Parametertypen in einer SUB/FUNCTION: Erfolgt nicht über den DIM-Befehl, sondern direkt in der Parameterliste mit 'AS <Typ>' (Ist in PowerBasic nur mit Variablen möglich, die mit SHARED deklariert sind). Beispiel:

```
DECLARE SUB Upro (anna AS LONG)       'Deklaration der SUB
CALL Upro(otto&)...                   'Aufruf der SUB
SUB Upro (anna AS LONG): anna=anna^2 'Definition der SUB
```

- Parameterübergabe-Methoden 'Call by Reference' und 'Call by Value' {11/154}:
  - Call by Reference: Normalerweise werden die Parameter an eine SUB/FUNCTION 'by Reference' übergeben, d.h. die SUB/FUNCTION erhält einen Zeiger auf den Parameter und alle Werteänderungen, die die SUB/FUNCTION an den Parametern durchführt beeinflussen den Wert der Ursprungsvariablen! Dies kann bei großen Softwareprojekten zu Softwarefehlern führen, die nur schwer zu finden sind.
  - Call by Value: Werden die einzelnen Übergabeparameter jeweils in zusätzliche Extra-Klammern gesetzt, so erhält die SUB/FUNCTION nur den Wert, nicht die Adresse. Bei Werteänderungen legt die SUB/FUNCTION dann eine eigene Variable an und die Ursprungsvariable bleibt unverändert. Call by Value ist nur bei der Übergabe von Einzelvariablen möglich; Feldern können nur 'by Reference' übergeben werden.
    Beispiel:

    ```
      CALL Drehen ((wort$), (anzahl%))  'Es wird nur der Wert von wort%
                         ' und anzahl% an die SUB übergeben ==> Die SUB
                         ' kann die Ursprungsvariablen nicht verändern
    ```

- Deklaration globaler Variablen und Felder (siehe {11/122} u.GLOBLVAR.BAS):
  Es gibt zwei Varianten für die Deklaration globaler Variablen und Felder:
  - VARIANTE 1: Global-Deklaration im Hauptprogramm (Normalvariante)

    Variablen des Hauptprogramms, die auch in einer SUB zugänglich sein sollen, müssen im Hauptprogramm mit SHARED als Globalvariable deklariert werden. Dies ist die am häufigsten verwendete Variante.
    - Globaldeklaration (immer am Programmanfang! zwei Möglichkeiten):
      - DIM SHARED {<Variable1> | <Feld> [(<Dimensionierung>)]>} AS <Typ> [,<Variable2> ...] –
        mit expliziter Typzuweisung; das Schlüsselwort DIM muß bei PowerBasic weggelassen werden.
      - COMMON SHARED <Variable1> [<Variable2>,...] – für mit Typ-Suffix implizit deklarierte Variable, nicht für Variable, die per DIM-Befehl deklariert sind {6/178}. Felder müssen in diesem Falle mit leeren Klammern notiert werden; sie sind also dynamisch.
        Beispiel:

        ```
        COMMON SHARED wochentag$
        CALL WeekOfDay
        ...
        SUB WeekOfDay                'Die Variable ist auch in
        wochentag$=...               'der SUB zugreifbar
        ```

    - Vorteile:
      - Alle Globalvariablen sind im Hauptprogramm in übersichtlicher Form an zentraler Stelle aufgelistet.
      - Einfacheres Handling, weniger Programmieraufwand
      - Auch statische (fest dimensionierte) Felder und anwenderdefinierte Felder mit TYPE..END TYPE sind möglich
    - Nachteile {11/125}:
      - Globalvariable sind in allen SUBs/ FUNCTIONs zugreifbar, auch wenn sie dort gar nicht benötigt werden. Dies kann zum unbeabsichtigten Ändern von Variablen führen und die Fehlerträchtigkeit bei großen Softwareprojekten erhöhen.
  - VARIANTE 2: Global-Deklaration in der SUB/FUNCTION (Spezialvariante)

    Variablen einer SUB/FUNCTION, die auch im Hauptprogramm zugänglich sein

sollen, müssen in der SUB/FUNCTION mit SHARED als Globalvariable deklariert
werden (Umgekehrung von Variante 1; nur gelegentlich verwendet)
- Globaldeklaration:
  SHARED <Variable1> [()] [AS <Typ>] [, <Variable2> [()] [AS...] - Globale Felder sind in diesem Falle grundsätzlich immer dynamisch, d.h. sie dürfen nicht dimensioniert werden (leere Klammern). Der Feldindex darf seltsamerweise - außer bei PowerBasic - höchstens ca. 10 betragen, siehe GLOBLFLD.BAS).
  Sollen die Globalvariablen auch in anderen SUBs/FUNCTIONs verwendet werden, so sind sie dort ebenfalls mit SHARED zu deklarieren (ist im Hauptprogramm nicht erforderlich).
- Vorteile der Variante 2:
  - Variablen sind nur in denjenigen SUBs/FUNCTIONs bekannt, die sie auch benötigen: Dies kann in großen Softwareprojekten Fehler vermeiden helfen {11/125}.
  - Allgemeine verwendbare SUBs/FUNCTIONs, die in vielen Programmen einsetzbar sind, lassen sich einfacher in ein neues Programm einfügen, da die DIM SHARED-Deklaration im Hauptprogramm entfällt.
- Nachteile der Variante 2:
  - Sollen die globalen Variablen und Felder auch in anderen SUBs/FUNCTIONs verwendet werden, so müssen sie dort ebenfalls erneut mit SHARED deklariert werden (mehr Programmieraufwand {11/124f}).
  - Nur dynamische Felder (ohne Dimensionierung) möglich

- Rekursiver Aufruf von SUBs/ FUNCTIONS: Siehe {9/101}, {11/243+247} und SORT.BAS.

** zum Inhalt ***

*** Subroutinen (Unterprogramme; max Länge: 64 KBytes)**

****

Subroutine definieren {2/13}, {3/131}:

```
- SUB <Name der Subroutine> [(<Formalparameter 1>, <Formalparameter 2> ...)]...
     ... [STATIC]
     [<Deklaration lokaler Variablen, Felder und Konstanten>]
     <Befehle> | [EXIT SUB]     'vorzeitiger Aussprung durch EXIT SUB möglich
  END SUB
```

- [STATIC] erhält den Wert aller lokalen Variablen zwischen 2 Aufrufen der Subroutine {3/130}. Es läßt sich bei Bedarf auch nur ein Teil der lokalen Variablen und Felder individuell als STATIC deklarieren; siehe Beschreibung des Befehls STATIC im Abschnitt Allgemeines zu Subroutinen.. .

Subroutine aufrufen

```
- CALL <Name der Subroutine> [(<Aktualparameter 1>, <Aktualparameter 2> ...)]
    oder
  <Name der Subroutine>  [<Aktualparameter 1>, ...]  'ohne Klammern! diese Va-
                                 'riante wird von PowerBasic nicht unterstützt
```

Subroutine im aufrufenden Hauptprogramm deklarieren:

... und Geltungsbereich d.Variablen: Siehe Allgemeines zu Subroutinen...

** zum Inhalt ***

* **Funktionen** (Unterprogramme mit Rückgabewert; max Länge 58 KB) {2/13}, {3/124}

****

Funktion definieren:

```
- FUNCTION <Name d.Funktion[Typ-Suffix]> [(<Formalparam.1>,<Formalparam.2> ..)]
       ... [STATIC]
    [<Deklaration lokaler Variablen, Felder und Konstanten>]
    <Befehle> | EXIT FUNCTION  'vorzeitiger Aussprung mit EXIT FUNCTION möglich
    <Name d.Funktion> = <Ausdruck>    'Rückgabewert zuweisen; dies muß unbedingt
  END FUNCTION                       'direkt vor END SUB erfolgen; notfalls
                                     'Zwischenvariable einführen
```

- [STATIC] erhält den Wert aller lokalen Variablen zwischen 2 Aufrufen
  der Funktion {3/130}. Es läßt sich bei Bedarf auch nur ein Teil der lokalen
  Variablen und Felder individuell als STATIC deklarieren; siehe
  Beschreibung des Befehls STATIC im Abschnitt Allgemeines zu Subroutinen.. .

Funktion aufrufen:

```
x=<Name d.Funktion[Typ-Suffix]> [(<Aktualparameter 1>, <Aktualparameter 2>..)]
```

Der Aufruf darf nur in einer Wertzuweisung (rechts von einem Gleichheitszeichen)
oder in einem Ausdruck stehen (z.B. hinter einer PRINT-Anweisung).
Beispiel: Aufruf der Funktion otto$ mit Übergabeparameter 5 :   anna$ = otto$(5)

Funktion im aufrufenden Hauptprogramm deklarieren:

... und Geltungsbereich d.Variablen: siehe Allgemeines zu Subroutinen...

Funktionen mit mehr als einem Rückgabewert :

Die Variablen für die Rückgabewerte werden ebenfalls als Aktual- und Formal-
parameter in die Parameterliste eingetragen - wie die Übergabeparameter.
Diese Vorgehensweise gilt auch für Subroutinen. Siehe auch NIBBLES.BAS/SUB
CONFIG und {9/94+96} {6/182}.

*** zum Inhalt ***

* **Lokale Subroutinen (GOSUB)**

****

- Hinweise:
  - Die Verwendung lokaler Subroutinen wird normalerweise nicht empfohlen; sie
    dienen weitgehend der Kompatibilität zu BASICA und GW-BASIC.
  - Die Definition einer lokalen Subroutine ist auch innerhalb einer SUB
    möglich und hat den Vorteil, daß die aufrufende SUB einfacher in andere
    QBasic-Programme übertragbar ist.
  - Im Hauptprogramm kann die lokale Subroutine auch nach dem END-Befehl defi-
    niert werden.
  - Eine Parameterübergabe an eine lokale Subroutine ist nicht möglich

```
- Deklaration        : nicht erforderlich
- Definitionsbeispiel: Potenz:          'Übergabeparameter nicht möglich !
                         x = 2 ^ i
                       RETURN
- Aufrufbeispiel     : GOSUB Potenz
- Variablen          : Eine lokale Subroutine hat keine lokalen Variablen, son-
                       dern kennt alle Variablen des aufrufenden Programms und
                       umgekehrt
```

*** zum Inhalt ***

* **Lokale Funktionen (DEF FN...)** {9/88}

```
****************************************************************************
```

- Hinweise:
  - Die Verwendung lokaler Funktionen wird normalerweise nicht empfohlen; sie dienen weitgehend der Kompatibilität zu den älteren BASIC-Dialekten BASICA und GW-BASIC.
  - Eine lokale Funktion muß am Beginn des Hauptprogramms definiert werden. Eine Definition am Ende des Hauptprogramms und in SUBs/FUNCTIONs ist - im Gegensatz zur lokalen Subroutine - nicht möglich.
  - Eine lokale Funktion kann über EXIT DEF vorzeitig verlassen werden.
  - An eine lokale Subroutine lassen sich beliebig viele Parameter übergeben.

```
- Definitionsbeispiel 1: DEF FNpotenz! (basis!) = 2 ^basis!  'Name muß mit FN
                                               'beginnen; einzeilige Funktion
'beginnen
- Definitionsbeispiel 2: DEF FNpotenz! (basis!)   'Name muß mit FN beginnen
                           <Befehle> | EXIT DEF   'vorzeitiger Aussprung mit
                           FNpotenz! = 2 ^ basis! 'EXIT DEF möglich
                         END DEF
- Aufrufbeispiel       : PRINT FNpotenz! (x!)     'Vor dem Aufruf muß die De-
                                                  'finition erfolgt sein!
- Variablen            : Eine lokale Funktion hat keine lokalen Variablen,
                         sondern kennt alle Variablen des aufrufenden Programms
                         und umgekehrt
```

*** zum Inhalt ***

* **DOS-Befehl oder externes EXE-Programm/ BAT-Batchdatei aufrufen**

```
****************************************************************************
```

- SHELL ["<Anweisung>"] - gibt den String "Anweisung" am DOS-Prompt aus. Nach Beenden des DOS-Programms erfolgt ein Rücksprung zum QBASIC-Programm. Fehlt die 'Anweisung', so wird zum DOS-Betriebssystem gewechselt und bei Eingabe von "EXIT" erfolgt der Rücksprung zum QBasic-Programm.

```
- Beispiele: SHELL "calcul.exe"    |  SHELL "dir c:\"
             SHELL "COPY "+ Adatei$ +" " +Bdatei$  'Adatei nach Bdatei kopieren
             SHELL "cd >xx.txt"  'aktuellen Pfadnamen in Datei xx.txt schreiben
```

BASIC-DOS-Dateisystembefehle aufrufen

- Hinweise:
  - Platzhalter "*", "?" in den Pfadnamen sind erlaubt ('Wildcards').
  - Die Fehlerbearbeitung, z.B. bei nicht vorhandenen Dateien, muß von Hand ausprogrammiert werden; siehe Fehlerbehandlung im Abschnitt 'Dateibear-beitung - Allgemeines'. Daher ist der DOS-Befehlsaufruf über SHELL häufig günstiger.

```
- CHDIR <Pfadname$>    - Wechsel in ein anders Verzeichnis
- KILL  <Dateiname$>   - Datei löschen
- MKDIR <Pfadname$>    - ein neues Unterverzeichnis erstellen
- RMDIR <Pfadname$>    - ein Unterverzeichnis köschen
- NAME  <alter Name$> AS <neuer Name$> - Datei oder Verzeichnis umbenennen
- FILES [<Pfadname$>] - zeigt den Inhalt des aktuellen Verzeichnisses oder
                        eines angegebenen Pfades an.
```

//////////////////////////////// Für Profis //////////////////////////////////

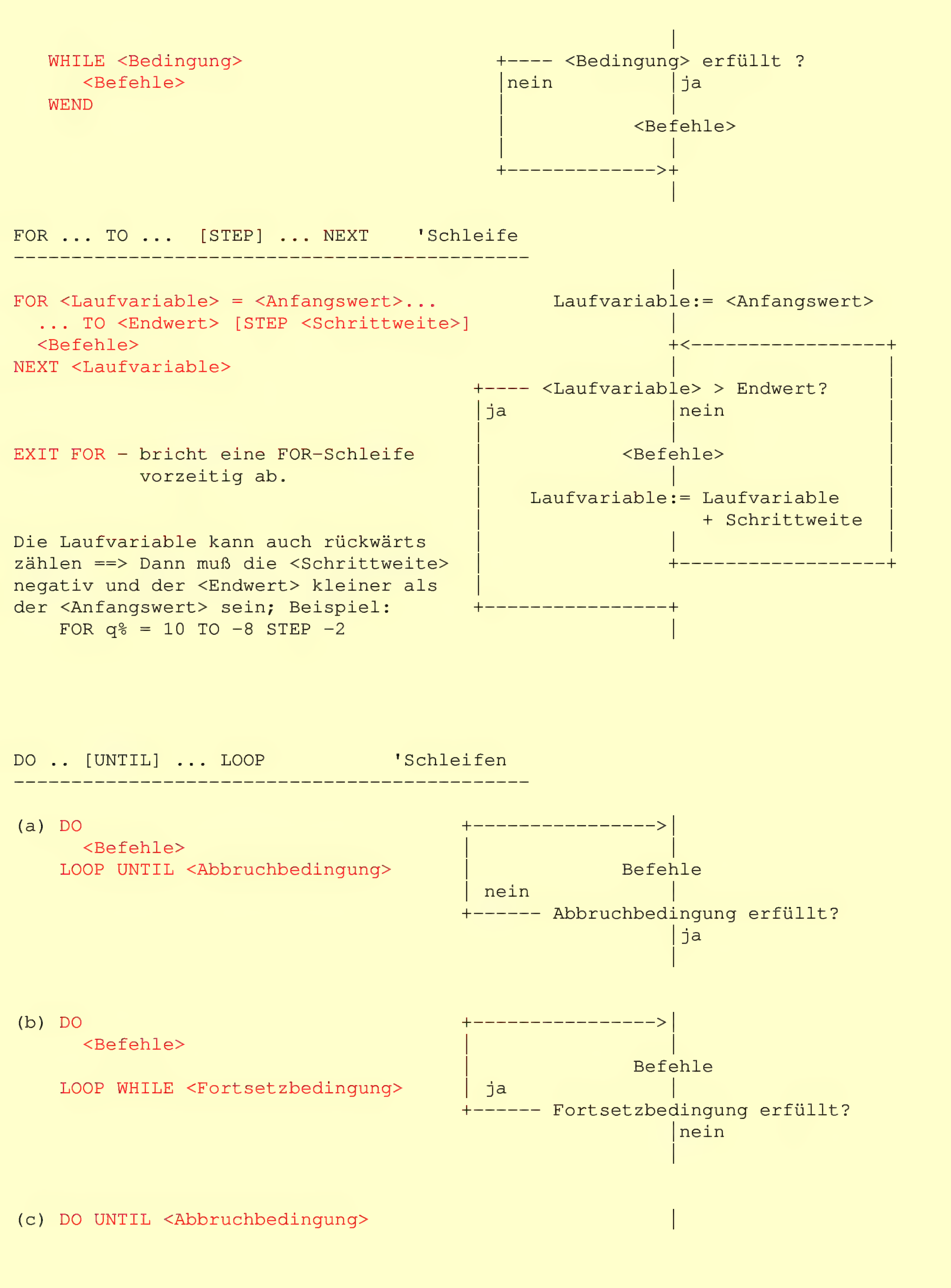
- DOS-Umgebungsvariable lesen und ändern {11/468}: Die in der AUTOEXEC.BAT gesetzten Umgebungsvariablen lassen sich durch ein QBasic-Programm ausle-sen und ändern. Änderungen bleiben jedoch nur während der Laufzeit des

```
    QBasic-Programms gültig
      - ENVIRON$ (<Nummer%>) - liefert den Setzbefehl der n-ten momentan
          gesetzten Umgebungsvariablen als Zeichenkette zurück.
          Beispiel:  FOR i = 1 TO 20: PRINT i; ENVIRON$(i): NEXT 'Anzeige der
                       'ersten 20 Umgebungsvariablen (mehr gibt es meist nicht)
      - ENVIRON$ (<NameDerUmgebungsvariablen$>) - liefert den in der AUTOEXEC.
          BAT stehenden Wert für die in Großbuchstaben angegebene Umgebungsva-
          riable als Zeichenkette zurück.
          Beispiele: PRINT ENVIRON$("PATH"), ENVIRON$("PROMPT")
                     PRINT ENVIRON$("BLASTER")
      - ENVIRON (<Name DerUmgebungsvariablen$> "=" {<Setzwert$> | <"">}) - Um-
         gebungsvariable setzen|löschen.
          Beispiele: ENVIRON "Path= C:\PROGIS"   'Pfad-Umgebungsvariable ändern
                     ENVIRON "PATH="             'Pfad-Umgebungsvariable löschen
```

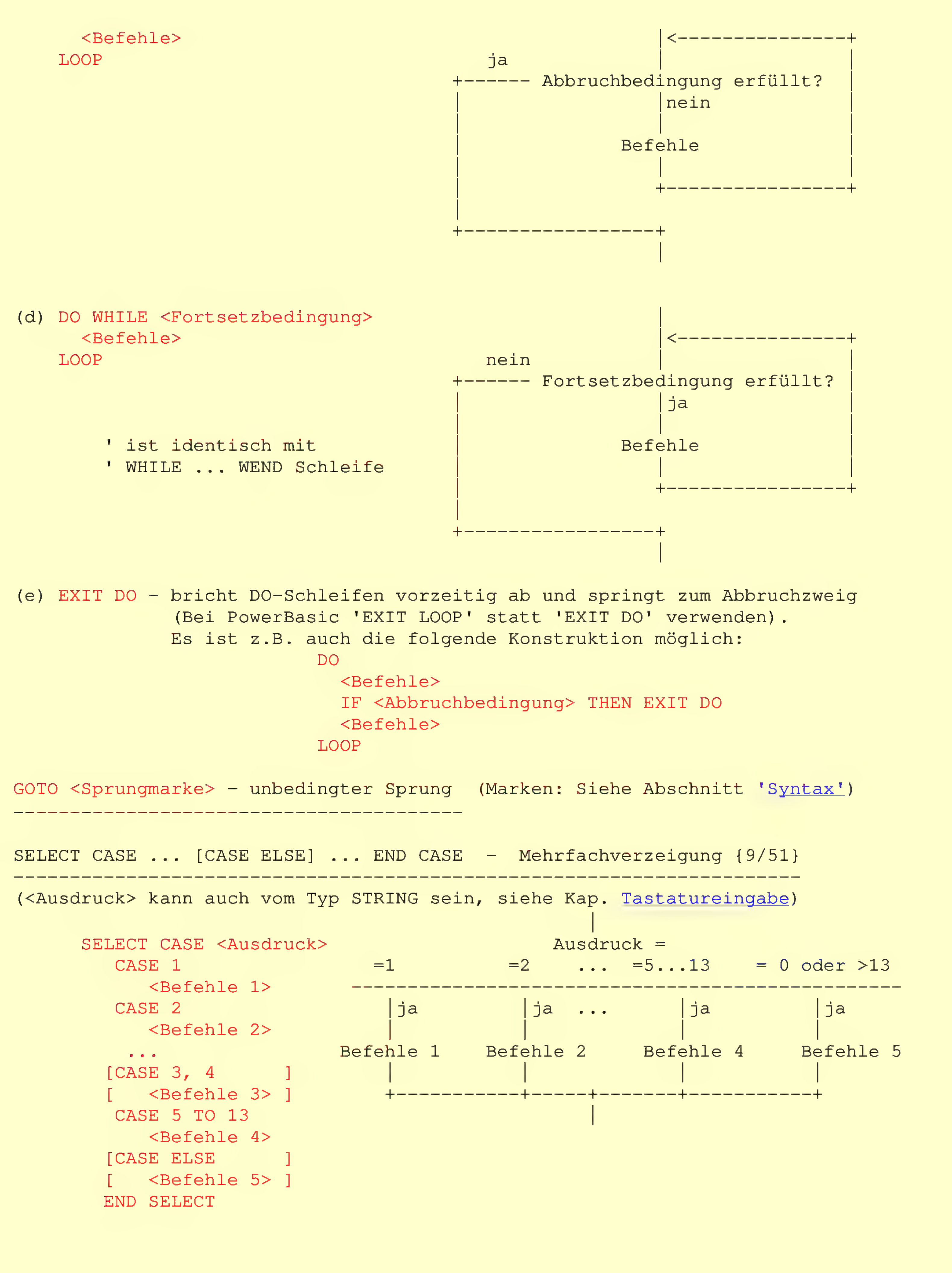
** zum Inhalt ***

*** Schleifen und Verzweigungen**

```
*******************************************************************************
IF ... THEN ... [ELSE]   'Verzweigung
-------------------------------------
Anmerkung: "Bedingung" wird als "erfüllt" (wahr/true) angesehen, wenn der
~~~~~~~~~~ Bedingungs-Ausdruck ungleich Null ist.

- Minimalversion in 1 Zeile (bei mehrzeiligem ELSE-Block muß die Normalversion
                             verwendet werden!):
     IF <Bedingung> THEN <Befehl> [ELSE <Befehl>]

- Normalversion mit IF-Block  [und ELSE-Block]
                                                           |
     IF <Bedingung> THEN                             Bedingung erfüllt?
        <Befehl 1>                                      |ja     |nein
        <Befehl 2>                                      |       |
    [ELSE          ]                                Befehl 1  Befehl 3
    [   <Befehl 3>]                                 Befehl 2  Befehl 4
    [   <Befehl 4>]                                     |       |
     END IF                                             +--+----+
                                                           |
- Mit Mehrfachverzweigung (ELSEIF im ELSE-ZWEIG)

     IF <Bedingung 1> THEN                                 |
        <Befehl 1> 'muß extra Zeile sein        Bedingung 1 erfüllt?
     ELSEIF <Bedingung 2> THEN                -----------------------------
        <Befehl 2>                                |ja                 |nein
     ELSE                                         |                   |
        <Befehl3>                                 |        Bedingung 2 erfüllt ?
     END IF                                       |         ------------------------
                                                  |           |ja           |nein
  Hinweis: Bei PowerBasic kann ein                |           |             |
  ~~~~~~~~ IF-Block mit EXIT IF               Befehl 1    Befehl 2     Befehl 3
           verlassen werden                       |           |             |
                                             --+-----------++------------+---
                                                           |

WHILE ... WEND                   'Schleife
---------------------------------------------
```



```
                                                         |
   WHILE <Bedingung>                      +---- <Bedingung> erfüllt ?
      <Befehle>                           |nein           |ja
   WEND                                   |               |
                                          |           <Befehle>
                                          |               |
                                          +------------->+
                                                         |

FOR ... TO ...  [STEP] ... NEXT    'Schleife
---------------------------------------------
                                                         |
FOR <Laufvariable> = <Anfangswert>...          Laufvariable:= <Anfangswert>
  ... TO <Endwert> [STEP <Schrittweite>]                 |
  <Befehle>                                              +<-----------------+
NEXT <Laufvariable>                                      |                  |
                                        +---- <Laufvariable> > Endwert?     |
                                        |ja              |nein              |
                                        |                |                  |
EXIT FOR - bricht eine FOR-Schleife     |            <Befehle>              |
           vorzeitig ab.                |                |                  |
                                        |    Laufvariable:= Laufvariable    |
                                        |                   + Schrittweite  |
Die Laufvariable kann auch rückwärts    |                |                  |
zählen ==> Dann muß die <Schrittweite>  |                +------------------+
negativ und der <Endwert> kleiner als   |
der <Anfangswert> sein; Beispiel:       +----------------+
    FOR q% = 10 TO -8 STEP -2                            |



DO .. [UNTIL] ... LOOP          'Schleifen
---------------------------------------------

(a) DO                                 +---------------->|
      <Befehle>                        |                 |
    LOOP UNTIL <Abbruchbedingung>      |             Befehle
                                       | nein            |
                                       +------ Abbruchbedingung erfüllt?
                                                         |ja
                                                         |


(b) DO                                 +---------------->|
      <Befehle>                        |                 |
                                       |              Befehle
    LOOP WHILE <Fortsetzbedingung>     | ja              |
                                       +------ Fortsetzbedingung erfüllt?
                                                         |nein
                                                         |


(c) DO UNTIL <Abbruchbedingung>                          |
```



```
      <Befehle>                                       |<---------------+
    LOOP                                   ja         |                |
                                        +------ Abbruchbedingung erfüllt? |
                                        |             |nein            |
                                        |             |                |
                                        |          Befehle             |
                                        |             |                |
                                        |             +----------------+
                                        |
                                        +-----------------+
                                                      |
```

```
                                                      |
(d) DO WHILE <Fortsetzbedingung>                      |<---------------+
      <Befehle>                            nein       |                |
    LOOP                                +------ Fortsetzbedingung erfüllt? |
                                        |             |ja              |
                                        |             |                |
        ' ist identisch mit             |          Befehle             |
        ' WHILE ... WEND Schleife       |             |                |
                                        |             +----------------+
                                        |
                                        +-----------------+
                                                      |
```

```
(e) EXIT DO - bricht DO-Schleifen vorzeitig ab und springt zum Abbruchzweig
             (Bei PowerBasic 'EXIT LOOP' statt 'EXIT DO' verwenden).
             Es ist z.B. auch die folgende Konstruktion möglich:
                         DO
                           <Befehle>
                           IF <Abbruchbedingung> THEN EXIT DO
                           <Befehle>
                         LOOP
```

GOTO <Sprungmarke> - unbedingter Sprung (Marken: Siehe Abschnitt 'Syntax')

SELECT CASE ... [CASE ELSE] ... END CASE - Mehrfachverzeigung {9/51}

(<Ausdruck> kann auch vom Typ STRING sein, siehe Kap. Tastatureingabe)

```
                                                    |
      SELECT CASE <Ausdruck>                    Ausdruck =
         CASE 1                 =1          =2    ...  =5...13    = 0 oder >13
            <Befehle 1>       -------------------------------------------------
         CASE 2                 |ja         |ja  ...     |ja         |ja
            <Befehle 2>         |           |            |           |
           ...               Befehle 1   Befehle 2    Befehle 4   Befehle 5
        [CASE 3, 4       ]      |           |            |           |
        [   <Befehle 3> ]       +-----------+-----+------+-----------+
         CASE 5 TO 13                             |
            <Befehle 4>
        [CASE ELSE       ]
        [   <Befehle 5> ]
        END SELECT
```

```
Kurzform:                   Sonderfall mit CASE IS {6/112}:
~~~~~~~~~                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    SELECT CASE i%              SELECT CASE i%
      CASE 1: x%=2                CASE IS < 23 : x=0    'i% < 23
      CASE 2: x%=5                CASE 50      : x=1    'i%=50
    END SELECT                    CASE IS > 127: x=3    'i% > 127
                                END SELECT
```

Hinweise zu PowerBasic:
~~~~~~~~~~~~~~~~~~~~~~~

- 'CASE IS' wird von PowerBasic nicht unterstützt.
- Ein SELECT CASE-Block kann bei PowerBasic mit EXIT SELECT vorzeitig verlassen werden.

*** zum Inhalt ***

* Modulare Programmierung und Bibliotheken

****

- CHAIN [Pfadname$] <Dateiname$> - übergibt die Kontrolle von dem aktuellen Programm an das BASIC-Programmmodul <Dateiname$> {11/454}. Eine automatische Rückkehr ins alte Programm findet nicht statt. Beispiel: CHAIN "C:\DOS\TEST.BAS"
- COMMON [SHARED] <Variablenliste> - Definition von Variablen und Feldern, die auch von anderen externen "gechainten" Programmoduln verwendbar sind. Die Reihenfolge der Variablen in der Variablenliste muß in beiden Programmen genau gleich sein! Bei Verwendung von SHARED sind die Variablen auch von allen SUBs/FUNCTIONs zugreifbar. {11/454}
- RUN [Pfadname$] <Dateiname$> - startet ein externes Programm. Wie CHAIN, jedoch werden vor dem Start des externen Programms alle Variablen gelöscht und alle offenen Dateien geschlossen. Beipiel: RUN "C:\DOS\TEST.BAS"
- In QuickBasic lassen sich über den QuickBasic-Linker Programme aus mehreren Dateien und Bibliotheken (*.LIB und *.QLB) zusammenbinden.
- In PowerBasic lassen sich über die Compiler-Anweisungen (Direktiven) $LIB und $LINK Bibliotheken sowie externe Dateien/ Units einbinden.

*** zum Inhalt ***

* Dateibearbeitung - Allgemeines, Dateiarten und Fehlerbehandlung {11/285ff}

****

Allgemeines zur Dateibearbeitung:
=================================

- Der Umgang mit Dateien ist in der einschlägigen Literatur und in der QBasic-Onlinehilfe nur bruchstückhaft und äußerst unsystematisch dargestellt. Viele Dinge findet man nur durch Probieren heraus. Diesem Mangel möchte das QBasic-Kochbuch abhelfen. Es behandelt daher die Dateizugriffe umfassend und in allen Facetten.
- Es lassen sich max. 255 Dateien beliebiger Größe mit jeweils max. 2 147 483 647 Datensätzen à max. 32 KB in einem Programm bearbeiten
- Jede offene Datei ist durch eine #Dateinummer% (#1...#255) gekennzeichnet.
- Max 16 Dateien dürfen gleichzeitig geöffnet sein.
- I/O-Geräte sind ebenfalls als Dateien definiert, z.B. "LPT1:" = Drucker, "SCRN:" = Monitor, "COM1:" = 1.serielle Schnittstelle/Maus, "KEYBD:" = Tastatur usw. {9/76} {11/304}.
- So kann man prüfen, ob eine Datei vorhanden ist:
  Variante 1: Bei sequentiellen Dateien, die zum Lesen geöffnet werden, Fehler "Datei nicht gefunden" abfragen entsprechend dem folgenden Bei-
~~~~~~~~~~~~~~~~~~~~~~~

```
spiel:          ON ERROR GOTO fehler
                OPEN "xxx.xxx" FOR INPUT AS #1
                GOTO weiter
                fehler: PRINT "Datei nicht vorhanden!"
                weiter: ...             'Datei vorhanden
```

Variante 2: Bei allen anderen Dateiarten: Dateilänge auf "0" abfragen:
IF LOF (<Dateinr. ohne #>) > 0 THEN ... 'Dateilänge > 0 ?
Funktioniert bei Dateien mit wahlfreiem Zugriff und binären Dateien. Bei sequentiellen Dateien nur in der Zugriffsart OUTPUT verwendbar.

Variante 3: Prüfen, ob ein definiertes Datum rücklesbar ist, siehe TOP96.BAS (bei Dateien mit wahlfreiem Zugriff). Wenn "0" ausgelesen wird, ist die Datei nicht vorhanden.

Welche Variante sinnvoll ist, hängt von der Dateiart und der Zugriffsart ab.

//////////////////////////////// Für Profis ////////////////////////////////

- Weitere Varianten der unten angegebenen OPEN-Befehle zum Öffnen einer Datei findet man in {11/303+311}, z.B. ACCESS READ WRITE: Öffnen zum Lesen und zum Schreiben
- Freigabe und Sperren von Dateien im Netzwerk: Siehe QBasic-Online-Hilfe unter OPEN, ACCESS, LOCK, UNLOCK sowie SHARED {11/303}
- FREEFILE - Die Funktion FREEFILE liefert die nächste noch freie, unbenutzte Dateinummer zurück {11/466}. FREEFILE kann von SUBs/FUNCTIONs genutzt werden, die nicht wissen können, welche Dateien bereits vom Hauptprogramm geöffnet sind.
- CLOSE (ohne Parameter) oder RESET schließt alle offenen Dateien.
- WIDTH (<#Dateinummer%>), <SpaltenZahl%> - legt die Spaltenzahl (Zeilenlänge) in einer Datei fest (wenig gebräuchlich {11/464})
- FILEATTR (<Dateinr>, {1|2} - liefert bei Attribut =1 den aktuellen Zugriffsmodus auf die Datei zurück (1=INPUT, 2=OUTPUT, 4=RANDOM, 8=APPEND, 32=Binary) und liefert bei Attribut=2 die - wenig interessante - DOS-Dateinr. zurück {11/467}.
- Mit BSAVE/BLOAD lassen sich Daten zwischen einem absolut adressierten Speicherbereich (z.B. Bildschirmspeicher) und einer Datei transferieren; siehe Abschnitt 'Speicherbereich mit BSAVE/BLOAD in Datei schreiben...'.

Arten von Dateien und Kriterien zur Auswahl der richtigen Dateiart:

Es gibt die folgenden 4 Dateiarten:
- Sequentielle Dateien (gebräuchlichste Dateiart)
- Dateien mit wahlfreiem Zugriff und TYPE-Puffer (häufig verwendet)
- Dateien mit wahlfreiem Zugriff und FIELD-Puffer (weniger gebräuchlich)
- Binäre Dateien (weniger gebräuchlich)

Diese Dateiarten unterscheiden sich in der Datenorganisation und den Zugriffsmechanismen. Die Dateiarten werden im folgenden kurz mit ihren Vor- und Nachteilen sowie ihren typischen Anwendungsschwerpunkten vorgestellt. Anschliessend ist jede Dateiart in einem eigenen Kapitel ausführliche beschrieben.

Sequentielle Dateien

- Sequentielle Dateien dienen zur Speicherung von Datensätzen, die grundsätzlich aus Textstrings (ASCII-Zeichen) bestehen. Numerische Werte werden ebenfalls als Strings abgelegt, ähnlich wie bei der Bildschirmausgabe über PRINT {6/336}.
- Ein Datensatz kann beliebig lang sein und endet mit Enter und Zeilenvor-

schub (CR + LF = CHR$(13) + CHR$(10) = Carriage Return + Linefeed)
- Ein Datensatz kann beliebig viele Felder enthalten, die beliebig lang sein können und durch Kommata voneinander getrennt sind. Die Datensätze können auch unterschiedliche Anzahl von Feldern beinhalten.
- Wahlfreier Zugriff auf einen beliebigen Datensatz ist nicht möglich. Es ist nur ein Zugriff auf aufeinanderfolgende Datensätze möglich ("inkrementierendes Lesen/ Schreiben") - beginnend am Dateianfang und endend am Dateiende.
- Vorteile :
  - Einfaches Handling, gebräuchlichste Dateiart neben der Datei mit wahlfreiem Zugriff und Type-Puffer
  - Datensätze können unterschiedlich lang und unterschiedlich strukturiert sein (spart u.U. viel Speicherplatz)
  - Ideal für kleine Dateien, die leicht in den Arbeitsspeicher hineinpassen, z.B. INI-Dateien und Highscore-Listen. Bei Highscore-Listen ist jedoch wegen der textbasierten Darstellung eine Manipulation durch Unbefugte mit einem beliebigen Editor leicht möglich!
- Nachteile:
  - Nur Texte speicherbar, numerische Größen werden ebenfalls als Text abgelegt und benötigen daher mehr Speicheplatz, z.B. INTEGER-Wert 4711 ==> "4711" (4 statt 2 Bytes).
  - Kein wahlfreier Zugriff, z.B. sind 25 Lesezugriffe erforderlich. um auf den 25.Datensatz einer Datei zuzugreifen!
  - Soll ein universeller Zugriff auf die Datensätze möglich sein, so muß die Datei zunächst 'am Stück' in ein RAM-Feld eingelesen und nach der Bearbeitung komplett wieder in der Datei gesichert werden.
  - Die Anzahl der in einer Datei gespeicherten Datensätze ist unbekannt.
  - Die genaue Position eines bestimmten Datensatzes innerhalb der Datei ist unbekannt.

Dateien mit wahlfreiem Zugriff und TYPE-Puffer

- Die Datei enthält Datensätze fester Länge, auf die beliebig über eine Datensatznummer zugegriffen werden kann.
- Ein Datensatz kann aus beliebig vielen Feldern beliebigen Datentyps, jedoch fester Länge bestehen. Alle Datensätze müssen gleich viele und gleich lange Felder haben. Ein bestimmtes Feld muß in allen Datensätzen den gleichen Datentyp haben.
- Als Schreib-/ Lesepuffer wird normalerweise ein mehrdimensionales Feld mit anwenderspezifischem Typ (TYPE...END TYPE) verwendet (siehe Abschnitt Felder unter ...Verbundfeld...).
- Vorteile :
  - Es muß nur der gerade benötigte Datensatz in den RAM-Speicher gelesen werden. Die restlichen Datensätze können in der Datei verbleiben - ideal für große Dateien.
  - Bequemer Zugriff auf beliebige Datensätzen über Datensatznummer
  - Flexible Datensatzstruktur mit Teil-Feldern beliebigen Typs
  - neben der sequentiellen Datei die gebräuchlichste Dateiart, gut geeignet für die Bearbeitung einer Datenbank mit fester Struktur
- Nachteile:
  - Datensätze und deren Teilfelder haben im Gegensatz zur sequentiellen Datei immer eine konstante Länge ==> u.U. hoher Speicherplatzbedarf.
  - Weniger geeignet für Datenbankprogramme, mit denen viele unterschiedliche Datenbank-Dateien angelegt und hantiert werden sol-

  len; Hierfür ist die Dateiart mit FIELD-Puffer besser geeignet.
  - Wird von früheren PowerBasic-Versionen nicht unterstützt (diese kennen keine TYPE...END TYPE Deklaration; bei V3.5 jedoch vorhanden).

Dateien mit wahlfreiem Zugriff und FIELD-Puffer (weniger gebräuchlich)

- Die Zugriffstechniken sind nahezu identisch mit den vorgenannten Dateien mit TYPE-Puffer. Jedoch erfolgt das Schreiben/Lesen nicht über einen anwenderspezifischen Datentyp (mehrdimensionales Feld), sondern über einen speziellen FIELD-Puffer, der nur einen Datensatz aufnehmen kann.
- Der FIELD-Puffer enthält ausschließlich Strings und kann in beliebig viele Teilfelder unterteilt sein. Numerische Werte müssen in 'Pseudostrings' umgewandelt werden
- Die Datensatzlänge und damit die Länge des FIELD-Puffer für eine konkrete Datei muß grundsätzlich immer gleich lang sein.
- Dieser Dateityp wird laut Microsoft weitgehend nur zur Kompatibilität mit den alten BASIC-Sprachen BASICA und GW-BASIC noch unterstützt.
- Vorteile gegenüber Dateien mit TYPE-Puffer:
  - Die Datenstruktur des FIELD-Puffers ist auch noch zur Laufzeit beliebig änderbar. Daher ist dieser Dateityp ideal geeignet zur Entwicklung regelrechter Datenbankprogramme, die unterschiedlichste Datenbankdateien anlegen und bearbeiten können {9/139ff}.
  - Auch bei PowerBasic verwendbar.

  Nachteile gegenüber Dateien mit TYPE-Puffer:
  - Etwas umständliche Hantierung
  - Numerische Werte müsen vor dem Schreiben in "Pseudostrings" umgewandelt und nach dem Auslesen aus der Datei wieder entsprechend in numerische Werte rückgewandelt werden.

Binäre Dateien (weniger gebräuchlich)

- Binäre Dateien sind quasi Byte-Felder ohne besondere Datensatzstruktur. Die Datenzugriffe erfolgen über einen Dateizeiger an beliebiger Stelle oder fortlaufend.
- Vorteile : Flexibelste Dateiart
- Nachteile: Der Programmierer muß sämtliche Strukturen, Dateizeiger und Dateninterpretationen selbst ausprogrammieren.

Fehlerbehandlung {3/181} {9/127} {6/358} {11/351}:

- Die Fehlercodes findet man in {9/135} und in der QBasic-Hilfe unter <Hilfe|Inhalt|Laufzeit-Fehlercodes>
- Fehlerroutine aufrufen, die unter <Marke2$> definiert ist:

```
ON ERROR GOTO <Marke2$>  'muß vor dem Fehler-verursachenden Befehl stehen
<Marke1$>:     ....      'hier kommen die Fehler-verursachenden Befehle
<Marke2$>:              'Beispiele für Fehlerbearbeitungen:
 IF ERR = 11|7 THEN.. 'Division durch 0 | zuwenig Speicherplatz
 IF ERR = 53 THEN ... 'Datei nicht gefunden
 IF ERR = 61|72 THEN..'Disk voll | defekt
 IF ERR = 64|76 THEN..'Dateiname unzulässig  | Pfad nicht gefunden
 IF ERR = 71 THEN ... 'Festplatte/ Diskette nicht bereit
 [RESUME]             'an der Zeile mit der Fehlerstelle fortsetzen
 [RESUME NEXT]        'an der Zeile hinter der Fehlerstelle fortsetzen
```

```
     [RESUME <Marke1$>]    'hinter Marke1$ fortsetzen
- Fehler simulieren (für Testzwecke): ERROR <fehlernr%>  {11/357}
- Auslösung anwenderdefinierter Fehler (Fehlernummern über 100; weniger
   gebräuchlich): z.B. IF zahl% = 1 THEN ERROR 111         {11/358}
- ERDEV  - liefert den DOS-Fehlercode zurück   {11/358}
- ERDEV$ - liefert den Namen des Fehler-verursachenden Geräts zurück, z.B.
   "A:" bei Fehler des Diskettenlaufwerks   {11/358}
- ERL    - liefert die Nummer der Fehler-verursachenden Programmzeile zurück,
   falls die Zeilen im Programm numeriert sind, sonst '0'   {11/359}
- ON ERROR GOTO 0 - Fehlerkontrolle abschalten (funktioniert nicht bei schweren
   Fehlern wie Division durch '0') {11/359}
```

*** zum Inhalt ***

* **Sequentielle Dateien bearbeiten** {9/109} {1/67} {3/145} {6/290}

****

```
- Allg. Hinweise:
   - Siehe auch Abschnitt 'Dateibearbeitung - Allgemeines..' und FILE-SEQ.BAS
   - In einer sequentiellen Datei lassen sich Informationen beliebiger Daten-
     typen ablegen. Text wird immer in Anführungszeichen abgespeichert, außer
     beim Schreiben der Datensätze mit dem PRINT-Befehl{11/305}. Numerische
     Werte erscheinen als abdruckbare ASCII-Zeichen.
     Beispiel: OPEN "XXX" FOR OUTPUT AS #1
               T$ = "Anna": x%=4711
               WRITE #1, T$, x%
            ==> Die Datei xxx enthält den folgenden Text:
                "ANNA",4711 <CR+LF> . Man beachte, daß der String 'ANNA' in
                Anführungszeichen in der Datei abgelegt ist (belegt 6 statt 4
                Zeichen).
   - Die Anzahl der in einer sequentiellen gespeicherten Datensätze ist prin-
     zipiell unbekannt. Sie muß gegebenenfalls in einer zweiten Datei abgelegt
     werden.
- Zugriffsarten: Der Dateizugriff ist über die folgenden Zugriffsarten möglich:
      - OUTPUT ==> Datensätze werden ab Dateibeginn fortlaufend geschrieben
                   (mit WRITE, PRINT oder PRINT USING). VORSICHT: Der alte
                   Dateiinhalt wird beim Öffnen einer Datei in der Zugriffsart
                   OUTPUT gelöscht!!!
      - APPEND ==> Datensätze werden hinten angefügt {1/71} (alter Dateiinhalt
                   wird beim Öffnen nicht gelöscht.
      - INPUT  ==> Datensätze werden ab Dateibeginn gelesen (mit INPUT oder
                   LINE INPUT)
- OPEN [Pfadname$] <Dateiname$> FOR {OUTPUT|INPUT|APPEND) AS #<Dateinr.1...255>
      - Öffnen einer Datei in einer der oben genannten Zugriffsarten.
      - Bei OUTPUT und INPUT wird der Dateizeiger auf den ersten Datensatz
        gesetzt.
      - Bei APPEND wird der Dateizeiger hinter den letzen in der Datei ge-
        speicherten Datensatz gesetzt.
      - Beim Öffnen einer nicht vorhandenen Datei in der Zugriffsart INPUT wird
        das Programm mit Fehler abgebrochen. Dies kann mit ON ERROR GOTO... ab-
        gefangen werden; siehe 'Fehlerbeandlung' im Abschnitt 'Dateibear-
        beitung Allgemeines...' .
- CLOSE [#<Dateinr.>]
      - Schließen einer Datei; vor einem Wechsel der Zugriffsart (durch
        OPEN...FOR) muß die Datei wieder geschlossen werden.
      - Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- Schreiben und Lesen von strukturierten Datensätzen (u.U. mit Teilfeldern):
```

- WRITE #<Dateinr.>, <Variable1> [,<Variable2>...,<Variable n>]
  - Schreiben eines Datensatzes [bestehend aus mehreren Teilfeldern] aus Variable(n) in eine Datei. Zwischen den Teilfeldern werden Kommas, hinter dem Datensatz ein <CR+LF> eingefügt. Strings werden in "Anführungszeichen" abgelegt
  - Die Daten werden an der aktuellen Dateizeigerposition in die Datei eingefügt {6/298}. Der Dateizeiger wird anschließend inkrementiert, d.h. auf den nächsten zu lesenden Datensatz gesetzt.
  - Die Datei muß vorher einmal in der Zugriffsart OUTPUT oder APPEND geöffnet worden sein.
- INPUT #<Dateinr.>, <Variable1> [,<Variable2>...,<Variable n>] {1/67}
  - Lesen eines Datensatzes [bestehend aus mehreren durch Kommas getrennte Teilfeldern] aus einer Datei in die Variable(n).
    Ist nur Variable1 angegeben, so erfolgt das Lesen nur bis zum ersten Komma.
  - Die Daten werden an der aktuellen Dateizeigerposition aus der Datei gelesen {6/298}. Der Dateizeiger wird anschließend inkrementiert, d.h. auf den nächsten zu lesenden Datensatz gesetzt.
  - Die Datei muß vorher einmal in der Zugriffsart INPUT geöffnet worden sein.

- Schreiben und Lesen von unstrukturierten, nicht in Teilfelder unterteilten Strings, die auch Kommas enthalten können (weniger gebräuchliche Alternative zu WRITE und INPUT):
  - PRINT #<Dateinr.> [USING <Maske$>] <Text$> [;|,]
    - Schreiben eines Text-Strings in eine Datei {6/294}; Syntax des PRINT-Befehls ist identisch mit der im Abschnitt 'Textausgabe auf Bildschirm' unter PRINT geschilderten Syntax für Bildschirmausgaben.
    - Die Daten werden an der aktuellen Dateizeigerposition ohne Anführungszeichen in die Datei eingefügt; dahinter wird <CR+LF> eingetragen. Der Dateizeiger wird anschließend inkrementiert, d.h auf die nächste Datensatz-Einfügestelle gesetzt.
    - Die Datei muß vorher einmal in der Zugriffsart OUTPUT oder APPEND geöffnet worden sein.
  - LINE INPUT #<Dateinr.>, <Stringvariable$>
    - Lesen eines Datensatzes (bis zum nächsten Enter= CR+LF) aus der Datei in die Stringvariable(n). Der Datensatz wurde ursprünglich typischerweise mit PRINT abgespeichert.
    - Die Daten werden an der aktuellen Dateizeigerposition aus der Datei gelesen. Der Dateizeiger wird anschließend inkrementiert, d.h. auf die nächste Datensatz-Lesestelle gesetzt.
    - Die Datei muß vorher einmal in der Zugriffsart INPUT geöffnet worden sein.
  - <Stringvariable$> = INPUT$ (<AnzahlZeichen%>), <Dateinr. ohne #>
    - Lesen einer wählbaren Anzahl von Zeichen (inklusive Kommas und CR+LF) aus der Datei in eine Stringvariable {9/114} {7/51} {11/306}, anschließend den Dateizeiger inkrementieren
- EOF (Dateinr. ohne #) - Funktion, liefert True (-1) zurück, nachdem der letzte Datensatz gelesen wurde.
- LOF (<Dateinr. ohne #>) - Funktion, liefert die Anzahl der in der Datei gespeicherten Bytes zurück (max 2^31-1).
- Befehle zum Bearbeiten des Dateizeigers (bei sequentiellen Dateien nicht besonders hilfreich, da der Dateizeiger Byte- und nicht Datensatz-orientiert ist 11/465}):
  - LOC (<Dateinr>) - liefert die aktuelle Byte-Position des zuletzt gelesenen oder geschriebenen Datensatzes geteilt durch 128

- SEEK (<Dateinr>) - liefert die Byte-Position des nächsten zu lesenden oder zu schreibenden Datensatzes zurück (1. Byte in der Datei hat die Nummer '1')
- SEEK <Dateinr>, <Position$> - setzt den Dateizeiger für den nächsten zu lesenden oder zu schreibenden Datensatzes auf die angegebene Byte-Position.

- Beispiel: Schreiben und Lesen von 2 Datensätzen, die in je 2 Felder unter-teilt sind:

```
OPEN "birth.dat" FOR OUTPUT AS #1 'existiert die Datei birth.dat
WRITE #1, "Thomas", "28.1.47"      'schon, so wird der Inhalt gelöscht!
WRITE #1, "Marlies", "29.02.49"
CLOSE #1
OPEN "birth.dat" FOR INPUT AS #1
FOR i=1 TO 2: INPUT #1, name(i), birthday(i): NEXT i
CLOSE #1
```

** zum Inhalt ***

*** Dateien mit wahlfreiem Zugriff und TYPE-Puffer bearbeiten** {9/118+123}

****

- Allg. Hinweise:
  - Siehe auch Abschnitt 'Dateibearbeitung - Allgemeines..' sowie {1/73} und TOP46.BAS.
  - Eine Datei mit wahlfreiem Zugriff besteht aus Datensätzen fester Länge, die in Felder ebenfalls fester Länge unterteilt sein können. Die Felder können sich im Datentyp und in der Länge voneinander unterscheiden. Die Datensätze lassen sich über ihre jeweilige Datensatznummer (ab 1) an-sprechen.
  - Als Zwischenpuffer für die aus der Datei gelesenen und in die Datei ge-schriebenen Datensätze dient ein anwenderdefiniertes Feld, das mit TYPE.. ..END TYPE deklariert werden kann (siehe unten).
  - Strings, die kürzer sind als deklariert, werden beim Schreiben rechts mit Blanks aufgefüllt, die nach dem Lesen mit RTRIM$ wieder beseitigt werden können.
- TYPE <Name des Typs> <Elementname1> AS <Typ> [<Elementname2> AS <Typ>] ... END TYPE
  - Anwenderdefinierten Datentyp deklarieren (mehrdimensionales Feld ge-mischten Datentyps; s. Abschnitt 'Felder'; muß im Hauptprogramm stehen).
- DIM <Feldname$> (<Feldlänge%> AS <Name des Typs>
  - gemischtes Feld deklarieren; kann auch in einer SUB/ FUNCTION stehen; sie-he Abschnitt 'Felder'.
- OPEN [Pfadname$] <Dateiname$> FOR RANDOM AS #<Dateinr. 1...255> LEN = <An-zahl Bytes je Datensatz> - Datei mit wahlfreiem Zugriff öffnen
- PUT #<Dateinr.>, <Datensatznr&>, <Feldname$> (<Nr% des Feldelements%>)
  - Datensatz mit der <Datensatznr.> aus RAM-Feld (anwenderdefiniertes ge-mischtes Feld) in die Datei schreiben.
- CLOSE [#<Dateinr.>]
  - Datei schließen; muß bei jedem Wechsel zwischen PUT und GET erfolgen. Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- GET #<Dateinr.>, <Datensatznr&>, <Feldname$> (<Nr% des Feldelements%>)
  - Datensatz mit der <Datensatznr.> aus der Datei ins RAM-Feld (anwenderde-finiertes gemischtes Feld) einlesen.
- LOF (<Dateinr. ohne #>) - Funktion, liefert die Anzahl der in der Datei ge-speicherten Bytes zurück (max 2^31-1).
- EOF - End-Of-File-Funktion funktioniert bei Dateien mit wahlfreiem Zugriff nicht!

- Befehle zum Bearbeiten des Dateizeigers {11/465}:
  - LOC (<Dateinr>) - liefert die Nummer des zuletzt gelesenen oder geschriebenen Datensatzes zurück
  - SEEK (<Dateinr>) - liefert den aktuellen Inhalt des Dateizeigers zurück, d.h. die Nummer des nächsten zu lesenden bzw. zu schreibenden Datensatzes
  - SEEK <Dateinr>, <Datensatznr$> - setzt den Dateizeiger für den nächsten Schreib-/ Lesevorgang auf die angegebene Datensatznummer.
- Beispiel: Einen gemischten Datensatz in die Datei "meinquiz.dat" schreiben und wieder rücklesen (siehe auch TOP46.BAS):

```
TYPE quiz                   'Datentyp "quiz" deklarieren: Feld m. je
   frage AS STRING * 70     '3 String-Elementen (70, 50 und 50 Zei-
   antw1 AS STRING * 50     'chen lang) und einem Integer-Element,
   antw2 AS STRING * 50     '(2 Bytes) ==> in Summe 172 Bytes
   oknr  AS INTEGER         'Deklaration muß im Hauptprogramm stehen!
END TYPE
DIM geschichte (1 TO 20) AS quiz
                            'Geschichtsquiz-Feld v.Typ "quiz" mit 20
                            'Feldelementen deklarieren (auch in SUB oder
                            ' FUNCTION möglich)
OPEN "meinquiz.dat" FOR RANDOM AS #1 LEN = 172 'Datei öffnen
PUT #1, 13, geschichte(13) '13. Element aus dem Feld geschichte in den
                            '13.Datensatz der Datei meinquiz.dat
                            'transferieren
CLOSE #1                    'Datei schließen
GET #1, 13, geschichte(13) '13. Element aus der Datei ins Feld geschich-
                            'te einlesen
CLOSE #1                    'Datei schließen
```

** zum Inhalt ***

* **Dateien mit wahlfreiem Zugriff und FIELD-Puffer bearbeiten** {9/118+139}{11/343}

****

- Allgemeine Hinweise:
  - Siehe auch Abschnitt 'Dateibearbeitung - Allgemeines..', FILE-FLD.BAS sowie {5/65}.
  - Vor dem Schreiben in die Datei muß ein Datensatz mit dem speziellen LSET-Befehl in den FIELD-Puffer eingetragen werden (siehe unten).
  - Ein Datensatz besteht aus einem oder mehren Field-Elementen (Datensatzfeldern)
  - Die enorme Flexibilität dieser Dateiart liegt darin, daß die Längen und Namen der Fieldelemente im FIELD-Puffer noch zur Laufzeit beliebig manipuliert werden können, so daß ein Anlegen und Bearbeiten beliebiger Datenbankstrukturen möglich ist, die zum Zeitpnkt der Programmentwicklung noch garnicht bekannt sein müssen {9/139}. Dies ändert jedoch nichts an der Tatsache, daß eine einmal geöffnete konkrete Datenbankdatei mit wahlfreiem Zugriff grundsätzlich nur gleich lange Datensätze speichern kann!
- OPEN [Pfadname$] <Dateiname$> FOR RANDOM AS #<Dateinr. 1...255> LEN = <Anzahl Bytes je Datensatz>
  - Datei mit wahlfreiem Zugriff öffnen
- FIELD #<Dateinr.>, <Field-Elementlänge1%> AS <Field-Elementname1$> [<Field-Elementlänge2%> AS <Field-Element-Name2$>]...
  - FIELD-Puffer für einen Datensatz deklarieren, u.U. bestehend aus mehreren Field-Elementen (Längen in Bytes). Ein Field-Element kann nur Strings enthalten. Numerische Werte müssen vor ihrem Eintrag in den FIELD-Puffer

mit einem MKx$-Befehl (s.u.) in einen "Pseudostrings" umgewandelt werden. Das Rückwandeln in numerische Werte nach dem Lesen erfolgt über einen entsprechenden CVx-Befehl (s.u.).

- {MKI$|MKL$|MKS|MKD$} (numerische Variable)
  - Aus INTEGER|LONG|SINGLE|DOUBLE-Variable Pseudostrings gleicher Länge erzeugen, die in einen FIELD-Puffer eingetragen werden können (Intel-Format: Lo- vor Hi-Byte)
- LSET <Field-Elementname$> = <Variable$>
  - In ein Field-Element innerhalb eines FIELD-Puffers eine Variable eintragen (Wertzuweisung). Überschüssige Zeichen werden rechts abgeschnitten, kurze Strings rechts mit Blanks aufgefüllt.
- RSET <Field-Elementname$> = <Variable$>
  - wie LSET, jedoch rechtsbündige Anordnung: Überschüssige Zeichen werden links abgeschnitten; weniger gebräuchliche Variante.
- PUT #<Dateinr.>, <Datensatznr.>
  - Datensatz aus dem FIELD-Puffer in den Datensatz mit der <Datensatznr.> in die Datei schreiben.
- CLOSE [#<Dateinr.>]
  - Datei schließen, muß bei jedem Wechsel zwischen PUT und GET erfolgen Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- GET #<Dateinr.>, <Datensatznr.>
  - Inhalt des Datensatz mit der <Datensatznr.> aus der Datei in den FIELD-Puffer transferieren.
- {CVI|CVL|CVS|CVD} (<String 2...8 Bytes>)
  - Pseudostring aus einem gelesenen FIELD-Puffer wieder in numerische Werte rückwandeln.
- <Variable> = <Field-Elementname$>
  - gelesenes Datensatz-Field-Element aus dem FIELD-Puffer lesen und in eine Variable eintragen. Beispiele:
    - anna$ = feld1$ 'Stringvariable
    - otto% = CVI(feld2$) 'numer. Variable, muß vorher rückgewandelt werden
- LOF (<Dateinr. ohne #>)
  - Funktion, liefert die Anzahl der in der Datei gespeicherten Bytes zurück (max 2^31-1).
- EOF - End-Of-File-Funktion funktioniert bei Dateien mit wahlfreiem Zugriff nicht!
- Befehle zum Bearbeiten des Dateizeigers {11/465}:
  - LOC (<Dateinr.>) - liefert die Nummer des zuletzt gelesenen oder geschriebenen Datensatzes zurück
  - SEEK (<Dateinr.>) - liefert den aktuellen Inhalt des Dateizeigers zurück, d.h. die Nummer des nächsten zu lesenden bzw. zu schreibenden Datensatzes
  - SEEK <Dateinr.>, <Datensatznr$> - setzt den Dateizeiger für den nächsten Schreib-/ Lesevorgang auf die angegebene Datensatznummer.
- Beispiel: (siehe FILE-FLD.BAS und {11/343}):

```
'Bearbeitung einer Telefon-Datenbank: Ein Name (Textstring) und eine Telefon-
'nummer werden zunächst in einen FIELD-Puffer eingetragen und dann von dort
'in den dritten Datensatz der Datenbank-Datei "Telefon" transferiert:
 OPEN "telefon" FOR RANDOM AS #1 LEN = 20  'Telefon-Datenbank, Länge 12+8=20
 FIELD #1, 12 AS name$, 8 AS no$  'FIELD-Puffer für 1 Datensatz deklarieren
                                  'mit 12 Bytes für Namen und 8 Bytes für
                                  'Telefonnummer
 LSET name$ = "antoni"            'Namen in FIELD-Puffer eintragen
 LSET no$ = MKL$(23852)           'Telefonnummer (LONG Integer) in
                                  'String wandeln u.in FIELD eintragen
```

```
    PUT #1, 3                       'Inhalt des FIELD-Puffers in den 3. Daten-
    CLOSE #1                        'satz der Datei schreiben
    OPEN "telefon" FOR RANDOM AS #1 LEN = 20  'wie oben, Datei öffnen zum Lesen
    FIELD #1, 12 AS name$, 8 AS no$  'wie oben; muß nochmals deklariert werden!
    GET #1, 3                       '3. Datensatz in den FIELD-Puffer lesen
    PRINT name$, "Telefon-Nr,"; CVL(no$)    'Inhalt des FIELD-Puffers anzeigen:
                        'numerischen 'Pseudostring' no$ vorher rückwandeln)
```

** zum Inhalt ***

* **Binäre Dateien bearbeiten** {5/65} {11/301}

****

- Allg. Hinweise:
    - Siehe auch Abschnitt 'Dateibearbeitung - Allgemeines..' und FILE-BIN.BAS
    - Binäre Dateien sind quasi Byte-Felder ohne besondere Datensatzstruktur. Die Datenzugriffe erfolgen über einen Dateizeiger an beliebiger Position oder fortlaufend. Das erste Byte hat die Position 1, das letzte Byte die Position LOF(<Dateinr.>)
- OPEN [Pfadname$] <Dateiname$> FOR BINARY AS #<Dateinr. 1...255>
  Binäre Datei zum Lesen und/oder Schreiben öffnen und Dateizeiger auf 1 setzen (d.h. aufs erste Byte in der Datei). Beim Wechsel zwischen Lesen und Schreiben (PUT und GET) muß die Datei nicht geschlossen werden.
- PUT #<Dateinr.>, [<Position>], <Variable>
  Variable ab der aktuellen Position des Dateizeigers [bzw. ab der angegebenen Position] in die Datei hineinschreiben und anschließend den Dateizeiger hinter das letzte geschriebene Byte setzen.
  Textvariable müssen vorher über DIM text AS STRING * <Länge> mit der richtigen festen Länge deklariert werden
- GET #<Dateinr.>, [<Position>], <Variable>
  Daten ab der aktuellen Position des Dateizeigers [bzw. ab der angegebenen Position] in eine Variable lesen und anschließend den Dateizeiger hinter das letzte gelesene Byte setzen.
  Textvariable müssen vorher über DIM text AS STRING * <Länge> mit der richtigen festen Länge deklariert werden
- Befehle zum Bearbeiten des Dateizeigers {11/465}:
    - LOC (<Dateinr>) - liefert die Position ('Location') des zuletzt gelesenen oder geschriebenen Bytes zurück (max. 2^31 - 1; das erste Byte der Datei hat die Nummer '1')
    - SEEK (<Dateinr>) - liefert die Byte-Position des nächsten zu lesenden bzw. zu schreibenden Bytes zurück
    - SEEK <Dateinr.>, <Byteposition&>
        - Dateizeiger auf eine wählbare Byteposition setzen
- CLOSE [#<Dateinr.>] - Datei schließen
  Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- EOF (Dateinr. ohne #) - Funktion, liefert True (-1) zurück, nachdem das letzte Byte gelesen wurde.
- LOF (<Dateinr. ohne #>) - Funktion, liefert die Anzahl der in der Datei gespeicherten Bytes zurück (max 2^31-1).
- Beispiel: Hex-Zahl 4711h (=18193) ab Byte 33 in Datei yyy.bin hinterlegen und wieder auslesen (siehe auch FILE-BIN.BAS):

```
     OPEN "yyy.bin" FOR BINARY AS #3
     z& = &H4711
     SEEK #3, 33         'Dateizeiger auf das 33. Byte setzen
     PUT #3, , z&        'LONG-Integerzahl in Byte 33...36 der Datei schreiben
     GET #3, 33, y&: PRINT y&
     PRINT "Die Datei yyy.bin ist"; LOF(3); " Bytes lang"
```

```
    CLOSE #3
```

** zum Inhalt ***

*** Druckerausgabe**

****

Die Druckerausgabe funktioniert nur mit Druckern, die den ASCII-Zeichesatz mit deutschen Umlauten verstehen, z.B. bei Druckern mit IBM Proprinter-Emulation. Der Windows-Standard-Druckertreiber wird in der DOS-Box nicht unterstützt.

- LPRINT <Text$> [;|.] - Text auf Drucker ausgeben {11/287} in neuer Zeile bzw [direkt hinter dem letzen gedruckten Zeichen | am Beginn des nächsten 14--Spalten-Bereichs]. Die Syntax entspricht dem PRINT-Befehl für Bildschirm-ausgaben. Ein Öffnen des Druckers über OPEN ist beim LPRINT-Befehl nicht er-forderlich.
- OPEN "LPT<DruckerNr%>:" FOR OUTPUT AS #<Dateinr> - Drucker für Ausgabe öffnen; die Ausgabe des Textes erfolgt mit WRITE oder PRINT, das Schließen des Druk-kers mit CLOSE - wie bei einer sequentiellen Datei; siehe den entsprechenden Abschnitt sowie {11/375}.
- WIDTH LPRINT <Spaltenzahl> - legt die Länge der Ausgabezeilen fest {11/464}
- WIDTH "LPT<DruckerNr%>:", <Spaltenzahl> - dito; z.B. ªWIDTH "LPT1:", 72
- LPOS (<DruckerNr%> - liefert die Anzahl der Zeichen zurück, die nach dem letz-ten <CR> (=CHR$(13)) ausgegeben wurden {11/466}.

** zum Inhalt ***

*** Serielle Schnittstellen** {11/375}

****

- OPEN "COM<Nr%>: <Option1> <Option2> ..." AS #<Dateinr>

  oder

  OPEN "COM<Nr%>: <Optionen>" FOR <Modus> AS #<Dateinr> LEN =<Länge> - Serielle Schnittstelle als Datei <Dateinr> öffnen mit den folgenden Optionen (siehe auch Abschnitte 'Dateibearbeitung...'):
  - 75|110|150|300|600|1200|2400|4800|9600 - Bitrate in Bits/s (Baud)
  - ,{N|E|O} - kein|gerades|ungerades Paritätsbit (None|Equal|Odd Parity)
  - ,{4|5|6|7|8} - Anzahl der Datenbits (ohne Paritätsbit; Vorbesetzung=7)
  - ,{1|1.5|2} - Anzahl der Stop-Bits (Vorbesetzung=1)
  - ,{ASC|BIN} - Öffnen für ASCII- | binäre Datenübertragung
  - ,CD <AnzMillisec> - Wartezeit in Millisec für Steuersignal DCD (Data Carrier Detect) zur Erkennung der Verbindungsaufnahme.
  - ,CS <AnzMillisec> - Wartezeit in Millisec für Steuersignal CTS (Clear To Send) zum Signalisieren der Sendebereitschaft.
  - ,OP <AnzMillisec> - Wartezeit in Millisec für 'Open Com', bis die Ver-bindung hergestellt ist.
  - ,LF - zusätzlicher Zeilenvorschub <LF> (=Linefeed = CHR$(10)) nach Wagenrücklauf <CR> (= Carriage Return = CHR$(13)) senden.
  - ,{RB|TB} <AnzBytes%> - Größe des Empfangs- | Sendepuffers in Bytes festlegen (typisch z.B. 2048 Bytes)
  - ,RS - Signal von der RTS-Leitung (Request to Send) unterdrücken (dient zur Sendeanfrage)

  Beispiel für 'normale Konfiguration':

  ```
  OPEN "COM2: 300,N,8,1, CD0,CS0,DS0,OP0, RS,TB2048,RB2048" FOR RANDOM AS #1
  ```

  - Serial Port 2 öffnen mit 300 Baud, ohne Parity-Bit, mit 8 Datenbits und einem Stop-Bit, ohne Wartezeiten und Handshake, je 2048 Bytes für Sende- und Empfangspuffer.
    Hinweis: Das Kommunikationsprogramm muß auf dem Empfänger-PC zuerst gestartet werden!
- LOC <Dateinr%> - Funktion, die die Nummer des Datensatzes zurückliefert, der

gerade gesendet oder empfangen wird, bei binären Dateien die Nummer des aktuellen Bytes. Ist noch nichts empfangen, so wird '0' zurückgeliefert.
- ON COM (<Nr%> GOSUB <Marke> - Ereignisgesteuertes Anspringen einer lokalen Subroutine, wenn ein neues Zeichen empfangen wurde.
- COM {ON|OFF|STOP} - Ereignisverfolgung für serielle Schnittstelle aktivieren | deaktivieren | unterbrechen mit Speicherung
- WIDTH COM <Nr>: , <Spaltenzahl%> - legt die Länge von Text-Ausgabezeilen fest {11/464}

** zum Inhalt ***

* **Direkter Speicherzugriff und I/O-Port-Zugriff**<(a> {11/392}

****

Speichermodell der 8x86-Prozessoren (Segment- und Offsetadressen)

Die 8x86-Prozessoren kennen im unteren ('konventionellen') 1 MB-Speicherbereich leider keine lineare Adressierung, sondern der Adreßraum ist in 64 KB große Segmente aufgeteilt, zwischen denen über die Segmentadresse umgeschaltet werden muß. Die Bytes innerhalb eines Segments werden durch die Offsetadresse angesprochen. Die physikalische, auf dem externen Adressbus erscheinende Speicheradresse wird auf dem CPU-Chip hardwaremäßig aus der aktuellen Segment- und Offsetadresse gemäß der folgenden Gleichung gebildet:

```
        Physikalische Adresse = Segmentadresse * 16  +  Offsetadresse
             (0...2^20)           (0...2^16)              (0...2^16)
```

Ablage von QBasic-Variablen im Speicher

- Numerische Variable: werden direkt an der durch VARSEG und VARPTR abfragbaren Adresse abgelegt und zwar im 'Intel-Format': Low-Byte vor High-Byte und Low-Word vor High-Word. Siehe auch PEEKPOK1.BAS und {10/27}. Vorzeichenbehaftete Größen müssen vor und nach dem Speichern trickreich in Bytes umgewandelt werden (siehe untenstehendes Beispiel 1)
- Felder: Auf alle Felder (statische, dynamische und anwenderdefinierte) greift QBasic mittels spezieller Feld-Deskriptoren zu, die über PEEK und POKE nicht zugänglich sind {10/38}
- Statische Strings: Statische Strings sind Strings fester Länge, die mit ... AS STRING * <Länge> deklariert sind. QBasic legt statische Strings direkt an der durch VARSEG(string$) und VARPTR(string$) abfragbaren Speicherposition ab. Statische Strings haben keinen String-Deskriptor {11/404}.
- Dynamische Strings: Alle implizit, d.h. ohne '... AS STRING * <Länge>' deklarierten Strings, sind dynamische Strings.
  Ihre Länge kann sich zur Laufzeit ändern. Der Zugriff auf dynamische Strings ist nicht direkt, sondern nur auf dem Umweg über einen String-Deskriptor möglich. Die Speicheradresse des String-Deskriptors für text$ ist über VARSEG(text$) und VARPTR(text$) abfragbar (siehe {11/393}, 9/30) und das untenstehende Beispiel 2). Der String-Deskriptor besteht aus zwei INTEGER-Werten: Die ersten beiden Bytes enthalten die Länge, die letzten beiden Bytes die Offset-Adresse des Strings. Der String befindet sich (außer bei PowerBasic und Quick Basic) grundsätzlich immer in demselben Segment wie der Stringdeskriptor.
  Hinweis zu PowerBasic und QuickBasic: Dort lassen sich die Adressen beliebiger Strings über SADD bzw. STRSEG/STRPTR direkt abfragen.

Bestimmung der absoluten Adresse von Variablen mit VARSEG und VARPTR

- VARSEG (<Variablenname>) - Segmentadresse einer Variablen ermitteln (Wertebe-

reich 0-65535; u.U. in LONG-Größe einlesen, da vorzeichenlos)
- VARPTR (<Variablenname>) - Offsetadresse einer numerischen Variablen oder einer statischen Stringvariablen ermitteln (Wertebereich 0-65535) bzw. Offsetadresse des String-Deskriptors einer dynamischen Stringvariablen (siehe oben).
- VARPTR$ (<Befehlsstring$>) - Selten verwendete Funktion zur Ermittlung der Stringadresse eines Befehlsstrings für den PLAY oder DRAW Befehl. Der Befehlsstring kann somit über einen Pointer mit vorangehendem "X" übergeben werden (Beispiel: PLAY "X" + VARPTR(<Variable$>); siehe QBasic-Onlinehilfe.
- VARPTR$ (<Variablenname$>) - Selten verwendete Funktion zur Ermittlung des Typs und der Offsetadresse einer Variablen als String mit drei Zeichen (siehe {11/469} und VARPTR$.BAS):
  - 1.Zeichen = Typ der Variablen: CHR$(2|3|4|8|20) = INT|STRING|SINGLE|DOUBLE|LONG
  - 2. und 3. Zeichen = Offsetadresse der Variablen (bzw. des String-Deskriptors bei dynamischen Strings) als String: 2.Zeichen = CHR$(Lo-Address), 3.Zeichen = CHR$(Hi-Address)

Speicherbytes lesen und schreiben mit PEEK und POKE

- Hinweis: Der direkte Speicherzugriff ist bei QBasic grundsätzlich nur Byteweise möglich. Bei PowerBasic wird über PEEKI und PEEKL auch ein Zugriff auf INTEGER- und LONG-Größen unterstützt.
- DEF SEG = <Segmentadresse&> Festlegen der aktuellen Segmentadresse (0...65536) für die folgenden PEEK und POKE-Befehle zum Schreiben/ Lesen von Speicherbytes
- DEF SEG - Wenn die Segmentadresse weggelassen wird, setzt DEF SEG die Segmentadresse wieder auf das QBasic-Standard-Datensegment zurück.
- PEEK (<Offsetadresse&) - Lesen eines Speicherbytes: Funktion vom Typ INTEGER, die im Low-Byte den Inhalt des durch die angegebene Offsetadresse adressierten Speicherbytes im aktuellen Segment zurückliefert. Die aktuelle Segmentadresse läßt sich durch DEF SEG verändern (siehe oben).
- POKE <Offsetadresse&>, <Wert%> - Schreiben eines Speicherbytes: Das niederwertige Byte von Wert% wird in das durch die Offsetadresse& adressierte Speicherbyte im aktuellen Segment geschrieben. Die aktuelle Segmentadresse läßt sich durch DEF SEG verändern (siehe oben).

Beispiele für die obengenannten Befehle zum direkten Speicherzugriff

- Beispiel 1: Variable e% mit PEEK lesen, inkremetieren und mit POKE zurück-
  ~~~~~~~~~~~ schreiben (siehe auch PEEKPOK1.BAS):

```
    a%=-4711  'bezüglich CVI und MKI$: Siehe Abschnitt Dateien mit FIELD-Puffer
    segm& = VARSEG(a%)  'Segmentadresse von a%
    offs& = VARPTR(a%)  'Offsetadresse von a%
    DEF SEG = segm&     'aktuelles Segment:=Segment, in d.sich a% befindet
    b$ = CHR$(PEEK(offs&)) + CHR$(PEEK(offs& + 1))
                        'Lo-/Hi-Byte als 'Pseudostring' lesen (Trick!)
    c% = CVI(b$) + 1    'Pseudostring wieder in INTEGER-Wert wandeln u.inkrem.
   '---- Inkrementierten Wert in a% zurückspeichern per POKE und anzeigen ------
   d$ = MKI$(c%)        'inkrementierten Wert in Pseudostring umandeln; Trick!
   POKE offs&, c%                          'POKE speichert immer nur das Lo-Byte
   hibyte% = CVI(RIGHT$(d$, 1) + CHR$(0))  'Hi-Byte ins Lo-Byte schieben
   POKE offs& + 1, hibyte%                 'Hi-Byte speichern
   PRINT a%
   DEF SEG              'Standard-Datensegment wieder aktivieren
```
  ~~~~~~~~~~~

- Beispiel 2: Dynamischen String text$ über Deskriptor lesen, ändern u. zurück- ~~~~~~~~~~~ schreiben (siehe auch PEEKPOK2.BAS und {10/31}):

```
text$ = "A-Hörnchen"  'Textstring abspeichern
segm& = VARSEG(text$) 'Segmentadresse des Deskriptors von text$ ermitteln
offs& = VARPTR(text$) 'Offsetadresse des Deskriptors von text$ ermitteln
DEF SEG = segm&       'aktuelles Segment := Segment, in dem sich sowohl der
                      'String als auch der String-Deskriptor befindet
stringadr& = CLNG(PEEK(offs& + 3)) * 256 + PEEK(offs& + 2)
                      'Hi- und Lo-Byte der eigentlichen Stringadresse aus
                      'Byte 3 und 4 des String-Deskriptors lesen; CLNG kon-
                      'vertiert INTEGER zu LONG (vermeidet Überlauf)
ersteszeichen% = PEEK(stringadr&) + 1 '1.Zeichen d.Strings lesen u.
                                      'inkrementieren (aus "A" wird "B")
POKE stringadr&, ersteszeichen%       'geändertes 1. Zeichen zurückspeichern
DEF SEG                               'Standard-Datensegment wieder aktivieren
```

Speicherbereich mit BSAVE/BLOAD in Datei schreiben und aus Datei lesen {11/401}

- BSAVE <Dateiname$>, <Offsetadresse&>, <AnzahlBytes%> - Ab der angegebenen Offsetadresse eine wählbare Anzahl von Speicherbytes in eine Datei schreiben. Die Datei braucht nicht explizit geöffnet und geschlossen zu werden. Das aktuelle Segment ist über DEF SEG anwählbar (siehe oben).
- BLOAD <Dateiname$> [, <Offsetadresse&>] - Mit BSAVE gesicherte Speicherbytes aus der Datei lesen und wieder an der alten Stelle [bzw. an der angegebenen Offsetadresse] im Speicher ablegen. Die Datei braucht nicht explizit geöffnet und geschlossen zu werden.
- Beispiel für BSAVE/BLOAD: Inhalt des Farb-Textbildschirms SCREEN 0 in die Datei xxx.bld speichern und anschließend wieder restaurieren (siehe 11/400 und BSAVE1.BAS):

```
DEF SEG = &HB800         'Segmentadresse des Farbbildschirms= B8000 hex
LOCATE 12, 30: PRINT "Dies wird gerettet": SLEEP
BSAVE "xxx.bld", 0, 4000  'Bildschirminhalt 4KBytes sichern nach xxx.bld
CLS : PRINT "Nix mehr da!!": SLEEP
BLOAD "xxx.bld": SLEEP   'gesicherten Bildschirminhalt wieder restaurieren
DEF SEG                  'Standard-Datensegment wieder aktivieren
```

Externe Maschinenspracheprogramme aufrufen {11/402}

- CALL ABSOLUTE <Offsetadresse&> - Externes Maschinenspracheprogramm unter der angegebenen Offsetadresse aufrufen (Segmentadresse kann durch DEF SEG definiert werden; siehe oben).
- CALL ABSOLUTE (<Parameter 1>, <Parameter 2>, ... Offsetadresse&>) - wie oben, jedoch mit Übergabe von Parametern.
- Hinweise zu QuickBasic und PowerBasic: Über CALL INTERRUPT können System-Interrupt-Routinen direkt angesprungen werden. Bei Verwendung von CALL ABSOLUTE muß QuickBasic mit 'QB /L' aufgerufen werden, um die Quick-Library QB.QLB einzubinden.

Zugriff auf I/O-Ports {11/467}

- INP (<I/O-Adresse%>) - Byte von I/O-Port lesen (ähnlich PEEK)
- OUT <I/O-Adresse%>, <Wert%> - niederwertiges Byte von Wert% zum I/O-Port senden (ähnlich POKE); Beispiel: OUT &H42, LSB% 'Speaker-Port ansteuern, d.h. 'I/O-Adresse 42 Hex

- WAIT <I/O-Adresse%>, <AND-Bitmuster%> [,<XOR-Bitmuster%] - Programm solange
  anhalten bis am I/O-Port die Bitkombination des AND-Bitmusters erscheint
  [bzw. die mit dem XOR-Bitmuster Exklusiv-Oder-verknüpfte Bitkombination]

Zugriff auf Gerätetreiber {11/468}

- IOCTL$ (#<Dateinr.>) - Steuerzeichen (Statusdaten) von einem Gerätetreiber
  empfangen
- IOCTL #<Dateinr.>, <Steuerzeichenfolge$> - Steuerzeichen an einen Geräte-
  treiber senden

Vorhandenen freien Speicherplatz für Variablen und Stack abfragen und ändern
----
- FRE (0|-1|-2) - vorhandenen Speicherplatz für Stringvariablen|numerische Va-
  riablen|Stack rückmelden. Insgesamt stehen ca. 30 KB Speicherplatz für
  Strings zur Verfügung {11/251+279+282}.
- FREE("") - bewirkt ein Aufräumen des String-Speichers ("Garbage Collection")
  und kann eventuell zusätzlichen Speicher für Stringvariablen freigeben.
- CLEAR, , <AnzahlBytes> - Speicherplatz für den Stack in gewünschter
  Größe reservieren und initialisieren; Startwert für Stackgröße = 1200 Bytes.

*** zum Inhalt ***

Umstieg von QBasic nach MS QuickBasic V4.5 {11/482}

****

- Vorteile von QuickBasic gegenüber QBASIC:
  - echter Compiler, erstellt ausführbare EXE-Dateien
  - unterstützt Module und Bibliotheken, Quelltexte über '$Include einbindbar
  - einige zusätzliche Befehle (siehe unten)
- Portieren von QBasic-Programmen nach QuickBasic:
  QBasic-Programme sind problemlos auch unter QuickBasic ablauffähig und
  zu EXE-Dateien kompilierbar; bei Verwendung des CALL ABSOLUTE Befehls
  wird jedoch die Quick-Library QB.QLB benötigt, und QuickBasic muß über
  'QB /L' aufgerufen werden (ebenfalls erforderlich bei Verwendung von
  INTERRUPT[X] usw.) {9/6}
- Zusätzliche Befehle und Schlüsselwörter bei QuickBasic:
  - '$INCLUDE - Compiler-Anweisung zum Einfügen von Quelltext aus einer anderen Datei (Include-Datei)
  - ALIAS - Verweist auf den Namen einer 'Nicht-BASIC-Prozedur"
  - BYVAL - Bewirkt 'Call by Value' statt 'Call by Reference' für einen Parameter, der an eine Nicht-Basic-Prozedur übergeben wird
  - CDECL - Bewirkt die Parametrübergabe an eine Prozedur gemäß C-Kon- ventionen
  - CALLS - Aufruf von Subroutinen, die in anderen Programmierspra- chen geschrieben wurden (Nicht-Basic-Prozeduren)
  - COMMAND$ - Liefert die Befehlszeile zurück, mit der ein QickBasic-EXE- -Programm aufgerufen wwurde und ermöglicht so, Übergabe- parameter abzufragen (siehe 'Parameterübergabe' im Abschnitt 'Bedienung...)
  - LOCAL|SIGNAL- für künftige Anwendungen reservierte Schlüsselwörter
  - SADD - Offsetadresse einer Stringvariablen
  - INTERRUPT|INTERRUPTX|INT86|INT86X - direkter Systeminterrupt-Aufruf
  - SETMEM - Verändern des 'Far-Heap'-Speicherbereichs
  - UEVENT|ENENT- Anwenderdefinierte Ereignisverfolgung

*** zum Inhalt ***

* **Umstieg von QBasic nach PowerBasic V3.5** {11/485}

****

- Vorteile von PowerBasic gegenüber QBASIC:
    - echter Compiler, erstellt ausführbare EXE-Dateien
    - Built-In-Assembler vorhanden
    - unterstützt Module, Bibliotheken und Units
    - mehr Datentypen (BCD, erweiterte Genauigkeit, siehe Abschnitt 'Variable')
    - beliebig große dynamische Strings, huge Arrays)
    - EMS-Speicher-Support
    - indirekte Adressierung über Pointer möglich
    - Direktbearbeitung von Feldern (ARRAY SORT|SCAN...; siehe Abschn. 'Felder')
    - TSRs erstellbar (speicherresidente Programme)
    - höhere Geschwindigkeit (ca. 2* schneller als QuickBasic-EXE-Programme)
    - wesentlich mehr Befehle, z.B. MIN, MAX, ROUND, PEEKI für Integer-Zugriff, PEEKL für Long-Integer-Zugriff, PEEK$ und POKE$ für String-Zugriffe, Bit-Befehle usw.
    - Variablendeklarationen erzwingbar
    - _Under_score_ in Namen erlaubt
    - C-Bibliotheken lassen sich einbinden
    - Verschiedene Compiler-Optimierungs-Optionen wählbar (nach Geschwindigkeit oder nach Programmgröße)
- Portieren von QBasic-Programmen nach PowerBasic:
    - In DIM-Felddeklarationen 'TO' durch ':' ersetzen
    - 'DIM' und 'COMMON' vor SHARED-Anweisungen entfernen {11/279}
    - Nur INTEGER-Konstanten verwendbar. Bei diesen muß 'CONST' durch '%' ersetzt werden, z.B. %anz=37 statt CONST anz%=37. Andere Konstanten-Typen gibt es nicht.
    - Anwenderdefinierte Verbundfelder (Typendeklarationen TYPE ... END TYPE) sind erst ab V3.5 möglich und müssen bei älteren PowerBasic-Versionen entfernt werden. Statt dessen Einzeldeklarationen oder Flex-Strings verwenden (siehe Abschnitt 'Felder').
    - Bei CASE-Anweisungen eventuell vorhandenes 'IS' entfernen.
    - 'EXIT DO' durch 'EXIT LOOP' ersetzen
    - Subroutinen-Aufrufe mit CALL und Parameterklammern versehen.
    - DECLARE-Anweisungen für SUBs und FUNCTIONs die sich in derselben Datei befinden, im Hauptprogramm entfernen oder Parameterliste nur aus Typenbezeichnungen statt Namen zusammensetzen (z.B. SINGLE statt egon!).
    - SLEEP durch DELAY ersetzen bei Versionen PowerBasic-Versionen < V3.5
    - SCREEN 13 wird nicht direkt, sondern nur mit Spezial-Routinen bzw. Bibliotheken unterstützt.
    - Sprungmarken müssen in einer extra Zeile stehen.
    - Vor Abfrage der Joystick-Feuerknöpfe mit STRIG muß die Ereignisverfolgung durch STRIG ON aktiviert werden.

*** zum Inhalt ***

* **Hinweise zu bestimmten Programmierproblemen**

****

- Suchalgorithmen: siehe {11/237}
- Sortieren von Zeichenketten (alphanumerisch) und numerischen Feldern:
    - Shell Sort: {9/71ff}; {6/281ff}
    - Bubble Sort: {11/236} und SORT.BAS
    - Quick Sort (rekursiv): {11/241},{9/294} und SORT.BAS
    - Quick Sort (iterativ): {10} und QuickSort in QSUBFUN.BAS
- Kästen (auch abgerundete) auf den Bildschirm ausgeben: Siehe {3/41}, KAESTEN.BAS und Sub "BOX" in QSUBFUN.BAS. ASCII-Codes zum Zeichnen von Kästen:

```
           196 194                   205 203                      223
     218 +-------+--+ 191      201 +=======#==+ 187        219 ¤¤¤¤¤¤¤ 219
         ¦       ¦  ¦              #       #  #            219 ¤     ¤ 219
     179 ¦          ¦ 179      186 #          # 186        219 ¤¤¤¤¤¤¤ 219
         ¦     197 ¦               #     206 #                    220
     195 +-------+--¦ 180      204 #=======+==# 185    Schatten:  +-----+
         ¦       ¦  ¦              #       #  #                   ¦    ¦¦¦ 2*219
     192 +----------+ 217      200 +=======#==+ 188               +-----+¦¦ 2*219
           196 193                   205 202                       ¦¦¦¦¦¦¦ 2*219
                                                                      219
```
- Rundung von Zahlen: Siehe INT(x) im Abschnitt "Arithmetische Operatoren..."
- Von der Grafikkarte unterstützte Bildschirm-Modi (SCREENs) ermitteln: siehe {6/358} und SCREENS.BAS.
- Numerischen Wert in Binär-Ziffern-String umwandeln: Siehe {11/331}; in Power-Basic über BIN$ möglich.
- text$ zentriert ausgeben: LOCATE , 80- LEN(text$) / 2: PRINT text$
- Zugriff auf einzelne Bytes, z.B. zur Systemprogrammierung {11/399}:
```
        DIM byte AS STRING*1  'numerische Werte mit CHR$ und ASC wandeln
```
- Integer-Größe als vorzeichenlose Ganzzahl 0 ... 2^16-1 interpretieren (z.B. zur Systemprogrammierung):
```
  IF i%>=0 THEN UnsignedInteger& = i% ELSE UnsignedInteger& = i% + 64536
```
- Indirekte Adressierung von Variablen über Zeiger ('Pointer'):
  Die indirekte Adressierung wird von QBasic und QuickBasic nicht unter-stützt, nur von PowerBasic. Über folgenden Umwege läßt sich eine indirek-te Adressierung nachbilden:
  - Die Parameterübergabe an SUBs/FUNCTIONs erfolgt normalerweise durch Übergabe eines Zeigers auf den Parameter ('Call By Reference', siehe Abschnitt 'Parameter-Übergabemethoden').
  - Mit VARSEG und VARPTR läßt sich die Adresse einer Variablen ermitteln und mit PEEK/POKE ein - allerdings byteweiser - Lese-/Schreibzugriff auf diese Speicherzelle realisieren; siehe Beispiel 1 im Abschnitt 'Direk-ter Speicherzugriff...'.
- Bildschirminhalt einlesen: Siehe TOP46.BAS, {11/399}, SCREEN-Funktion im Ab-schnitt 'Textausgabe auf Bildschirm' sowie BSAVE/BLOAD im Abschnitt 'Direkter Speicherzugriff...'.
- Kurven mit Koordinatenkreuz anzeigen: Siehe {11/214} und SINUS.BAT.
- Elemente eines Variablenfeldes einfügen, löschen, sortieren, suchen: Siehe {11/253ff}; in PowerBasic durch die Befehle ARRAY {SORT|SCAN|INSERT|DELETE} direkt unterstützt {11/486}.
- Menüsystem einbauen: {11/415}
- Maus verwenden: {11/407}
- Wochentag zu einem bestimmten Datum ermitteln:
  Siehe Function 'WeekDay' in CLOCKFIX.BAS
- Aktuelle Cursorposition retten und restaurieren (rückretten):
```
   X=POS(0): Y=CSRLIN       'Cursorposition retten
   .... PRINT ...           'zwischenzeitliche Bildschirmausgeaben
   LOCATE Y,X               'Cursorposition restaurieren
```
  Dies ist z.B. in einem Interruptprogramm erforderlich, das Bildschirmausgaben durchführt, z.B. sekündliche Uhrzeitanzeige mit ON TIMER (1) GOTO... .

** zum Inhalt ***

* Internet-Links zu QBasic

****

Deutsche QBasic- und PowerBasic-Seiten:

==
  Q-Basic.de - Offizielle deutsche QBasic Homepage mit Compiler-Download
  PBsound - Größte Deutsche PowerBasic Seite von Thomas Gohel
  QBasic-Seite von C.Teubner
  MasterCreating - Spieleschmiede mit QBasic-Download-Perlen und Bibliotheken
  QBasicPlus - +++ QBasic mal ganz anders +++ mit Compiler-Download
  PKWORLD - Pawel's QuickBasic-Seite mit Compiler-Download
  Meff's PowerBasic-Seite mit gutem Tutorial für Sprite-Animation
  PowerSoft - QBasic-Page von P.Hanzik und S.Hölzel
  QuickBasic.Megapage.de - Die inoffizielle deutsche QuickBasic-WebSite!

Englische QBasic- und PowerBasic-Seiten:
===
  QBasic.com - alles zu und über QBasic
  PowerBASIC - Homepage des Herstellers eines rasend schnellen BASIC-Compilers
  ABC-Archiv - 3000 BASIC-Programme in Rubriken unterteilt mit Komfort-Browser
  ABC-Archiv - Europäischer Mirror des All Basic Archive
  Alternatelogic.QuickBasic.com - Tonnenweise QBasic-Tutorials in englisch
  QBasic Pathfinder - mit Compiler-Download und riesiger Linksammlung
  Future Software QB Programming - Board, viele Tools und Bibliotheken
  DirectQ - Eine der besten QuickBasic-Bibliotheken, mit Grafik/Sound
  QBASIC Download Libraries - Viele Beispielprogramme + Tools
  QBasicAmerica
  MK Molnar - Englische QBasic Seite
  NeoZones - Große englische Webseite zu QBasic und QuickBASIC
  Toshi's Project Page - Qbasic-Seite
  The QBasic Site
  Didi's Homepage - Englische QBasic Homepage
  TVP's QBasic Site - Free Programs and Source

Was kommt nach QBasic? - Seiten über BASIC-Sprachen für Windows:
===
  PROFAN² - Einfache, Basic-ähnliche Windows-Programmiersprache (Freeware)
  GFA-BASIC für DOS, Win31 und Win95
  XBasic - Mächtiger Freeware Basic-Compiler für Windows'95'98'NT
  Visual Basic 3 Kursscripts von CAMPS in Deutsch
  BasicWorld - Deutsche Zeitschrift für Visual BASIC-Programmierer
  Visual Basic - Online Magazine
  VBA Magazin - Fachmagazin für MS-Office- u. Datenbankentwickler
  Visual Basic Tips - von Guido Niester
  VB Programmer's Source - CodeGuru
  Visual Basic Utilities, Tools von SOFTSEEK

** zum Inhalt ***
* **Literatur zu QBasic** (Literaturhinweise: {x/n} = Seitennummer n im Buch {x})
****
Sämtliche aufgeführte Literatur war im Sommer 1999 im Buchhandel bzw. im Internet erhältlich - außer evtl. {11}. Wer ein Online-Buch im Internet nicht mehr findet, kann dies gern bei mir unter thomas.antoni@erlf.siemens.de per E-Mail anfordern.

{1} bestehend aus den folgenden 3 Online-Büchern:
(a) "QBasic lernen"
<== http//www.lookup.com/homepages/80948/qb
(b) Wir lernen QBasic", Kursskript von R.Jessenberger, ca. 25 Seiten
sehr gut. < == http://rzaix340.rz.uni-leipzig.de/~mai94cbg/sz1.html
(c) "QBasic-Tutorial" by Mallard, ca. 45 Seiten englisch, sehr gut
<== http://www.qbasic.com/tutorial.shtml
{2} "Dokumentation II" bestehend aus dem Online-Buch
"Skript zum QBasic-Kurs der Friedrich-Schiller-Realschule Böblingen"
von Wilfried Furrer, sehr guter Einführungskurs! ca. 40 Seiten,
zu finden unter "Online Angebot | Arbeitsgemeinschaften"
<== http://www.ba-stuttgart.de/~wfurrer/fsr-bb/main.htm
{3} "Das Einsteigerseminar QBASIC" von Heinz-Gerd Raymans, bhv-Verlag, 1998
ISBN 3-89360-672-6, 215 Seiten, 19,80 DM
{4} "Meine 15 schönsten Quick-BASIC Programme" Band 1, Ludwig Schulbuch,
ISBN 3-929466-58-9, 98 Seiten, 14,80 DM, sehr guter QBasic-Kurs für
Anfänger
{5} "Meine 15 schönsten Quick-Basic Programme" Band 2, Ludwig-Schulbuch,
ISBN 3-929466-61-9, 114 Seiten, 14,80 DM, sehr guter QBasic-Kurs für
Fortgeschrittene
{6} "Arbeiten mit QBASIC" von M.Halvorson, Vieweg-Verlag, ISBN 3-528-05164-7,
520 Seiten, ca. 88,- DM; vollständige, etwas trockene Einführung in
QBasic
{7} "Dokumentation VII " bestehend aus dem Online-Buch:
"Wir lernen QBasic - A new Method" von C.-D. Volko, ca. 70 Seiten,
<== http://www.q-basic.de
Sehr gutes Tutorial in deutsch!! Zu finden unter "Hilfe".
{8} "Dokumentation VIII": QBasic Tutorial and Language Quick Summary
{9} "Programming in QuickBASIC" von N.Kantaris, Bernard Babani Books,
ISBN 0 85934 229 8, 173 Seiten, englisch, ca. 25,-, sehr gut, ausführ-
liche Behandlung der Dateizugriffe und von Datenbank-Lösungen, jedoch
Grafik und Sound nicht behandelt
{10} Elektronisches Buch <== www.ethanwiner.com
"PC Magazine's BASIC Techniques and Utilities", sehr gutes
über 500 Seiten umfassendes, professionelles englisches Buch für fort-
geschrittene QuickBasic-Programmierer. gigantisch !!!!
{11} "Das QBasic 1.1 Buch" von H.-G. Schuhmann, Sybex-Verlag, 1993,
ISBN 3-8155-0081-8, 550 Seiten mit Diskette, 59,- DM, sehr gut, behandelt
fast alle QBasic-Befehle mit vielen Beispielen, beschreibt den Umstieg
auf QuickBasic und PowerBasic; evtl. nicht mehr erhältlich.
{12} "The Revolutionary Guide to QBasic" von V.Dyakonov u.a., Wrox Press Ltd.,
1996, ISBN 1-874416-20-6, 578 Seiten mit Diskette, 73,80 DM, gigantisch
gut, behandelt alle QBasic-Befehle und sehr ausführlich die Spiele-,
Sound- und 2D/3D-Grafikprogrammierung, auch mit Animationen. In leicht
verständlichem Englisch geschrieben.

zum Inhalt

----------------------- Ende des QBasic-Kochbuchs ---------------------